Devoir Surveillé d'ITC N°1

le 15 décembre 2023

MPSI & PCSI

Consignes

- Les codes doivent être présentés en mentionnant explicitement l'indentation, au moyen par exemple de barres verticales sur la gauche.
- Pour qu'un code soit compréhensible, il convient de choisir des noms de variables les plus explicites possibles.
- La numérotation des exercices (et des questions) doit être respectée et mise en évidence. Les résultats (hors questions purement informatiques) doivent être encadrés proprement.
- Il est important de numéroter correctement les pages des copies qui seront données à la correction. Chaque candidat est responsable de la vérification de son sujet d'épreuve : pagination et impression de chaque page.
- Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il convient de le signaler sur la copie et de poursuivre la composition en expliquant les raisons des initiatives qui ont été prises.
- Les candidats ne doivent avoir aucune communication entre eux ou avec l'extérieur durant l'épreuve. Aussi, l'utilisation des téléphones portables et, plus largement, de tout appareil permettant des échanges ou la consultation d'informations, est interdite.
- À l'issue de la durée prévue pour cette épreuve, les candidats doivent déposer le stylo et ne sont plus autorisés à écrire quoi que ce soit sur leur copie. Tout retard donne lieu à une pénalité sur la note finale.
- **L'usage de la calculatrice est interdit.**

Les signatures des fonctions devront toutes être écrites !

Problème **Simulation de la cinétique d'un gaz parfait** La théorie cinétique des gaz vise à expliquer le comportement macroscopique d'un gaz à partir des mouvements des particules qui le composent. Depuis la naissance de l'informatique, de nombreuses simulations numériques ont permis de retrouver les lois de comportement de différents modèles de gaz comme celui du gaz parfait.

Ce sujet s'intéresse à un gaz parfait monoatomique. Nous considèrerons que le gaz étudié est constitué de N particules sphériques, toutes identiques, de masse m et de rayon R, confinées dans un récipient rigide.

Les simulations seront réalisées dans un espace à une ou deux dimensions ; le récipient contenant le gaz sera, suivant le cas, un segment de longueur L ou un carré de côté L.

Dans le modèle du gaz parfait, les particules ne subissent aucune force (leur poids est négligé) ni aucune autre action à distance. Elle n'interagissent que par l'intermédiaire de chocs, avec une autre particule ou avec la paroi du récipient. Ces chocs sont toujours élastiques, c'est-à-dire que l'énergie cinétique totale est conservée.

Dans tout le sujet, on suppose que les bibliothèques `math` et `numpy` ont été importées grâce aux instructions :

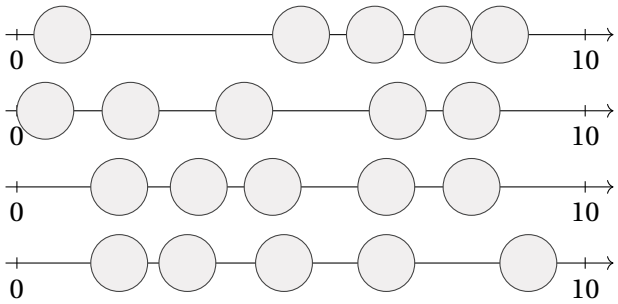
```
import math
import numpy as np
```

En particulier, on pourra utiliser la commande `np.random.rand()` qui retourne un réel choisi aléatoirement dans l'intervalle $[0, 1[$.

Partie I — Placement des particules en dimension 1 Pour pouvoir réaliser une simulation, il convient de disposer d'une situation initiale, c'est-à-dire d'un ensemble de particules réparties dans le récipient. L'objectif de cette partie est de positionner de manière aléatoire un ensemble de particules.

Dans toute cette partie, nous travaillerons en dimension 1 et nous nous intéresserons seulement à la position des particules, chaque particule étant représentée par l'abscisse de son centre.

L'objectif est donc de placer N particules (sphères de rayon R) le long d'un segment de longueur L sans qu'elles se chevauchent ni qu'elles sortent du segment. La figure ci-dessous montre quelques exemples de placements possibles avec $N = 5, R = 0,5$ et $L = 10$.



On décide d'écrire une fonction `placement1D` qui construit aléatoirement, à partir des paramètres géométriques du problème (nombre et rayon des particules, longueur du segment), une liste d'abscisses correspondant à la position initiale du centre de chaque particule.

1. On propose la fonction `test_bords` suivante :

```
def test_bords(x:float,L:float,R:float)->bool:
    if x < R or x > L-R:
        return False
    else:
        return True
```

Expliquer à quoi correspondent les paramètres fournis à cette fonction, et ce qu'elle retourne.

2. Écrire une fonction `test_nonchevauchement` à qui on fournit deux réels `x` et `y` (qui représentent les abscisses des centres de deux particules) et un réel `R` (qui représente le rayon commun des particules) et qui renvoie `True` si les particules de centres `x` et `y` ne se chevauchent pas, et `False` dans le cas contraire.

3. On considère la fonction `placement1D` suivante :

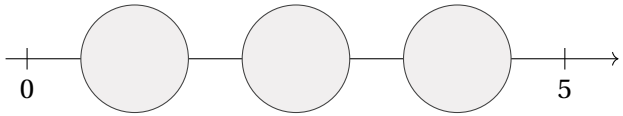
```
def placement1D(N:int,R:float,L:float)->list:
    Resultat = []
    while len(Resultat) < N:
        a = L*np.random.rand()
        test = True
        for p in Resultat:
            if test_nonchevauchement(a, p, R) == False:
                test = False
        if test and test_bords(a,L,R):
            Resultat.append(a)
    return Resultat
```

3.1) Détailler l'action de la ligne `a = L*np.random.rand()`.

3.2) Expliquer l'intérêt de la boucle `for`.

3.3) Que renvoie `placement1D(N,R,L)` ?

4. On considère l'appel `placement1D(4, 0.5, 5)` et on suppose que les trois premières particules ont été placées aux points d'abscisses 1, 2,5 et 4 (voir figure ci-dessous). Quelle sera la suite du déroulement de la fonction `placement1D` ?



Partie II — Optimisation du placement des particules en dimension 1

Pour placer aléatoirement N particules le long d'un segment, nous allons maintenant envisager une approche plus efficace que celle étudiée précédemment. L'idée est de calculer l'espace laissé libre sur le segment par les N particules une fois qu'elles sont positionnées puis de répartir aléatoirement cet espace libre entre les particules. Nous allons suivre les étapes suivantes :

1ère étape : déterminer ℓ, espace laissé libre par les N particules dans le segment $[0, L[$;

2ème étape : placer aléatoirement N particules virtuelles ponctuelles ($R = 0$) dans le segment $[0, \ell[$; à cette étape, deux particules peuvent parfaitement occuper la même abscisse : il n'y a pas de conflit possible ;

3ème étape : trier les abscisses des particules virtuelles par ordre croissant.

4ème étape : remplacer chaque particule virtuelle par une particule réelle de rayon R en décalant toutes les particules (réelles et virtuelles) situées plus à droite de façon à dégager l'espace nécessaire.

Reprenons le dernier exemple du paragraphe précédent, consistant à placer 4 particules de rayon 0.5 sur un segment de longueur $L = 5$. La première étape nous donnerait $\ell = 1$, la seconde étape nous donnerait les abscisses aléatoires :

`[0.02362, 0.57630, 0.16575, 0.49017]`
Ces abscisses seraient triées par la troisième étape pour obtenir
`[0.02362, 0.16575, 0.49017, 0.57630]`
et la quatrième étape donnerait finalement les abscisses définitives des particules :
`[0.52362, 1.66575, 2.99017, 4.07630]`.

5. En imaginant les N particules placées dans le segment $[0, L]$, déterminer une expression de l'espace ℓ laissé libre en fonction de L, R et N.

6. Écrire une fonction `placement_virtuel` à qui on fournit un réel ℓ strictement positif et un entier naturel N et qui renvoie une liste constituée de N nombres, éventuellement égaux, choisis aléatoirement entre 0 et ℓ.

7. Dans cette question, nous allons mettre en place le tri des abscisses des particules.

7.1) La fonction suivante est une première étape pour le tri d'une liste. Expliquer précisément ce qu'elle renvoie.

```
def mystere(L:list)->int:
    m, p, n = L[0], 0, len(L)
    for i in range(1, n):
        if L[i] > m:
            m, p = L[i], i
    return p
```

7.2) Recopier la fonction `tri_selection` suivante sur votre copie et la compléter afin qu'elle trie par ordre croissant la liste `L` fournie en paramètre par effet de bord grâce à la méthode de tri par sélection.

```
def tri_selection(L:list)->None:
    n = len(L)
    for i in range(...):
        p = mystere(L[:n-i])
        L[p], L[n-i-1] = ...
```

7.3) Appliquer cette méthode de tri sur la liste suivante, en donnant l'évolution de la liste à chaque étape de la boucle : `L = [5.31, 1.2, 10, 7.34, 8.4]`.

8. Notons $V(0), \ldots, V(N-1)$ les places virtuelles des N particules, et $R(0), \ldots, R(N-1)$ les places réelles. On a donc :

$$R(0) = V(0) + R, \quad R(i) - R(i-1) = V(i) - V(i-1) + 2R, \quad \forall i \in [\![1, N-1]\!].$$

8.1) Prouver que : $\forall i \in [\![0, N-1]\!], \quad R(i) = R + 2iR + V(i)$

8.2) Compléter la fonction `placement1Drapide` suivante à qui on fournit le nombre N de particules à placer, le rayon R commun de toutes ces particules et la longueur L du segment, et qui retourne la liste des abscisses des particules une fois placées.

```
def placement1Drapide(N:int , R:float , L:float) -> list:
    l = ....... # Etape 1
    res = []    # Etape 2
    for k in range(N):
        res.append(.......)
    tri_selection(res) # Etape 3
    for i in range(N): # Etape 4
        res[i] = ..............
    return res
```

PARTIE III — PLACEMENT DES PARTICULES EN DIMENSION 2

L'algorithme optimisé pour un segment, n'est pas utilisable pour un espace de dimension égale à 2. Nous allons donc généraliser la fonction `placement1D` pour la transformer en une fonction utilisable dans un espace de dimension 2.

Dans cette partie, nous ne tiendrons toujours pas compte de la vitesse des particules. Ces dernières seront représentées par une liste à deux éléments. Par exemple, `p = [1.23, 3.64]` désignera une particule dont le centre a pour abscisse 1.23 et pour ordonnée 3.64.

9. Écrire une fonction `distance2D` à qui on fournit deux listes `p1` et `p2` représentant les coordonnées des centres de deux particules, et qui renvoie la distance euclidienne entre les centres de ces deux particules.

10. Écrire une fonction `test_bords2D` à qui on fournit une liste à deux éléments `p` (qui représente les coordonnées du centre d'une particule), un réel `L` (qui représente la taille du carré $[0, L] \times [0, L]$ dans lequel on souhaite placer les particules), un réel `R` (qui représente le rayon de la particule) et qui renvoie **`True`** si la particule `p` est contenue toute entière dans le carré de côté L et **`False`** dans le cas contraire.

11. Écrire de même une fonction `test_nonchevauchement2D` à qui on fournit deux listes `p1` et `p2` (qui représentent les coordonnées des centres de deux particules) et un réel `R` (qui représente le rayon commun des particules) et qui renvoie **`True`** si les particules de centres `x` et `y` ne se chevauchent pas, et **`False`** dans le cas contraire.

12. Écrire enfin, en s'inspirant de la fonction `placement1D`, une fonction `placement2D` qui renvoie la liste des coordonnées des centres de N particules sphériques de rayon R placées aléatoirement dans un récipient carré de côté L dans un espace à 2 dimensions.

PARTIE IV — MOUVEMENT DES PARTICULES EN DIMENSION 1

Dans cette partie, on travaille à nouveau dans un espace de dimension 1, c'est-à-dire, que les particules évoluent sur un segment de longueur L. On suppose que chaque particule est représentée sous la forme `p = [x, v]` où x désigne l'abscisse (en mètres) du centre de la particule et v sa vitesse (en mètre par seconde). Par exemple, les variables :

```
p1 = [5.335, 412.3]
p2 = [3.1, 4.86]
```

représentent deux particules dont les centres respectifs ont pour abscisses 5.335 et 3.1, et les vitesses respectives sont $412.3 m.s^{-1}$ et $4.86 m.s^{-1}$.

On suppose que l'on dispose également d'une fonction `situationInitiale(N:int,R:float,L:float) -> list`: qui renvoie une liste de N particules de rayon R, chaque élément de la liste étant une 2-liste de la forme

expliquée précédemment, placées aléatoirement et correctement sur un segment de longueur L.

À partir de cette situation initiale, les positions et vitesses des particules vont évoluer au gré du déplacement des particules, des différents chocs entre elles et des rebonds sur les parois. On appelle *évènement* chaque choc ou rebond sur la paroi.

13. [Analyse physique] Considérons deux particules de masses m_1 et m_2 qui entrent en collision avec les vitesses initiales $\overrightarrow{v_1}$ et $\overrightarrow{v_2}$.
Les vitesses $\overrightarrow{v_1'}$ et $\overrightarrow{v_2'}$ des deux particules après le choc sont données par :

$$\begin{cases} \overrightarrow{v_1'} = \dfrac{m_1 - m_2}{m_1 + m_2}\overrightarrow{v_1} + \dfrac{2m_2}{m_1 + m_2}\overrightarrow{v_2} \\ \overrightarrow{v_2'} = \dfrac{2m_1}{m_1 + m_2}\overrightarrow{v_1} + \dfrac{m_2 - m_1}{m_1 + m_2}\overrightarrow{v_2} \end{cases}$$

13.1) Que deviennent ces formules lorsque $m_1 = m_2$? À quel évènement cela correspond-il ?

13.2) Que deviennent ces formules lorsque m_1 est très petite devant m_2 ? À quel évènement cela correspond-il ?

14. [Evolution des particules]

14.1) Pour une particule de vitesse constante v, on a l'égalité $x(t) = x(0) + t\,v$. En déduire la fonction `vol(p:list, t:float) -> None` : qui met à jour l'état de la particule `p` au bout d'un vol de t secondes sans choc ni rebond.

14.2) Écrire la fonction `rebond(p:list) -> None` : qui met à jour la vitesse de la particule `p` suite à un rebond sur l'un des deux bords du segment. *La fonction `rebond` n'est pas chargée de vérifier que la particule se trouve au contact d'une paroi.*

14.3) Écrire la fonction `choc(p1:list, p2:list) -> None` : qui modifie les vitesses des deux particules `p1` et `p2`, suite au choc de l'une contre l'autre. *La fonction `choc` n'est pas chargée de vérifier que les deux particules sont en contact.*

PARTIE V — INVENTAIRE DES ÉVÈNEMENTS EN DIMENSION 1 Chaque évènement sera représenté par une liste de trois éléments avec la signification suivante, indice par indice :

indice 0 : un flottant donnant le nombre de secondes, à partir de l'instant courant, au bout duquel l'évènement aura lieu ;

indice 1 : un entier compris entre 0 et N − 1 donnant l'indice dans la liste des N particules de la première (ou seule) particule concernée par l'évènement ;

indice : 2 un entier compris entre 0 et N − 1 donnant l'indice de la deuxième particule concernée par l'évènement ou `None` s'il n'y a pas de deuxième particule concernée (l'évènement correspondant est alors un rebond sur une paroi) ;

Ainsi, par exemple :

- `[0.4, 34, 57]` désigne le choc entre les particules d'indice 34 et 57 qui aura lieu dans 0,4 s.
- `[1.7, 34, None]` désigne le rebond de la particule d'indice 34 sur l'un des bords du segment qui aura lieu dans 1,7 s.

Dans cette partie, on suppose à nouveau que chaque particule est représentée sous la forme `p = [x, v]`.

15. [Prochains évènements]

15.1) On suppose que la particule étudiée va rencontrer une des parois du récipient, en faisant abstraction de toute autre particule qui pourrait se trouver sur son chemin. Selon le signe de la vitesse, la particule va heurter la paroi de droite ou la paroi de gauche du récipient. On déduit de l'égalité $x(t) = x(0) + t\,v$ que, si $v \neq 0$, $t = \dfrac{x(t) - x(0)}{v}$, ce qui nous conduit à discuter :

- Si $v < 0$, la particule va heurter la paroi de gauche et on aura alors :
$$t = \frac{\mathrm{R} - x(0)}{v}.$$
- Si $v > 0$, la particule va heurter la paroi de droite et on aura alors :
$$t = \frac{\mathrm{L} - \mathrm{R} - x(0)}{v}.$$

Écrire une fonction d'en-tête `def tr(p:list, R:float, L:float) -> None or float:` qui retourne le temps à attendre pour que la particule `p` de rayon R rencontre une paroi du récipient de longueur L. Cette fonction devra renvoyer `None` si la particule ne rencontre jamais de paroi.

15.2) Écrire une fonction d'en-tête `def tc(p1:list, p2:list, R:float) -> None or float:` qui détermine si les deux particules `p1` et `p2`, de rayon R, vont se rencontrer, en faisant abstraction de la présence des autres particules et des parois, autrement dit en considérant que ces deux particules sont seules dans un espace infini. Cette fonction renvoie `None` si les deux particules ne se rencontrent jamais, sinon elle renvoie le temps (en secondes) au bout duquel les particules entrent en collision.

16. [Catalogue d'évènements] On souhaite maintenant construire un catalogue des évènements qui pourraient se produire prochainement. Ce catalogue sera représenté par une liste dans laquelle les évènements, représentés par la liste de

trois éléments décrite au début de cette partie, sont ordonnés par date décroissante : le plus lointain en début de liste, le plus proche en fin de liste.

16.1) Écrire une fonction
`ajoutEv(catalogue:[[float, int, int or None]], e:[float, int, int or None]) -> None:`
qui ajoute au bon endroit dans la liste catalogue l'évènement `e`. La liste `catalogue` contient des évènements ordonnés par temps décroissant.

16.2) Compléter la fonction suivante qui ajoute, dans la liste ordonnée d'évènements `catalogue`, les prochains évènements potentiels concernant la particule d'indice `i` de la liste `particules` qui contient toutes les particules présentes dans le récipient. Le paramètre R donne le rayon d'une particule et L la longueur du récipient. Les évènements à prendre en compte sont le prochain rebond contre une paroi et le prochain choc avec chacune des autres particules. La fonction veillera à maintenir ordonnée la liste `catalogue`.

```
def ajout1p(catalogue:[[float, int, int or None]], i:int, \
↪ R:float, L:float, particules:[[list]]) -> None:
    for j in range(len(particules)):
        if j != i:
            t = tc(particules[i],particules[j],R)
            if t != ........:
                ........................
    t = ............
    if t != None:
        ........................
```

16.3) Écrire une fonction
`initCat(particules:[[list]], R:float, L:float) -> [[float, int, int or None]]:`
qui utilise la fonction `ajout1p` et qui renvoie la liste, ordonnée par temps décroissant, des prochains évènements potentiels concernant une liste de particules `particules` de rayon `R` dans un récipient de longueur `L`.

16.4) Expliquer pourquoi la liste renvoyée par la fonction `initCat` contient certains élèments qui correspondent en fait au même évènement.

Correction
du Devoir Surveillé d'ITC N°1
MPSI & PCSI

Solution

Partie I — Placement des particules en dimension 1

1. On fournit à la fonction `test_bords` un réel `x` (qui représente l'abscisse du centre d'une particule), un réel `L` (qui représente la longueur du récipient) et un réel `R` (qui représente le rayon de la particule). La fonction retourne **`True`** si la particule de centre `x` est contenue toute entière dans le segment $[0, L]$ et **`False`** dans le cas contraire.

2.
```
def test_nonchevauchement(x:float,y:float,R:float)->bool:
    if abs(x-y) < 2*R:
        return False
    else:
        return True
```

3. **3.1)** Cette ligne permet de stocker dans la variable a un réel choisi aléatoirement dans l'intervalle $[0, L[$.

3.2) La variable `p` parcourt tous les éléments déjà stockés dans la liste `Resultat`, et on s'assure que la nouvelle particule choisie ne vient pas chevaucher les particules déjà en place.

3.3) `placement1D(N,R,L)`, lorsqu'il se termine, va retourner une liste constituée des abscisses choisies aléatoirement des centres de N particules dans le récipient de longueur L et qui ne se chevauchent pas.

4. L'algorithme ne va jamais se terminer car il n'y a plus assez d'espace pour venir placer la quatrième particule. On a donc une boucle « infinie » !

Partie II — Optimisation du placement des particules en dimension 1

5. Chaque particule occupe un espace égal à 2R et il y a au total N particules, d'où $\boxed{\ell = L - 2NR}$.

6.
```
def placement_virtuel(l:float,N:int)->list:
    res = []
    for k in range(N):
        res.append(l*np.random.rand())
    return res
```

7. **7.1)** La fonction `mystere` retourne la position du premier élément maximal de la liste `L`.

7.2)
```
def tri_selection(L:list)->None:
    n = len(L)
    for i in range(n-1):
        p = mystere(L[:n-i])
        L[p], L[n-i-1] = L[n-i-1], L[p] # placement du \
        ↪ max en queue de liste
```

7.3) Voici l'évolution de la liste :
```
L = [5.31, 1.2, 10, 7.34, 8.4]
L = [5.31, 1.2, 8.4, 7.34, 10]
L = [5.31, 1.2, 7.34, 8.4, 10]
L = [5.31, 1.2, 7.34, 8.4, 10]
L = [1.2, 5.31, 7.34, 8.4, 10]
```

8. **8.1)** La suite $(R(i) - V(i))$ est arithmétique de raison 2R, donc, $R(i) - V(i) = 2iR + R(0) - V(0)$, soit encore $\boxed{R(i) = 2iR + R + V(i)}$. *Autre méthode : sommer la relation donnée entre les indices $i = 0$ et i*

8.2)
```
def placement1Drapide(N:int , R:float , L:float) -> list:
    l = L-2*N*R # Etape 1
    res = []    # Etape 2
    for k in range (N):
        res.append(l*np.random.rand())
    tri_selection(res) # Etape 3
    for i in range(N): # Etape 4
        res[i] = R+res[i]+2*i*R
    return res
```
```
>>> placement1Drapide(5, 0.5, 10)
[1.212785821339731, 4.310253826362095, 5.690360825930564, \
↪ 6.738105940202908, 7.912991781121075]
```

Partie III — Placement des particules en dimension 2

9.
```
def distance(p1:list , p2:list) -> float:
    S = (p2[0]-p1[0])**2 + (p2[1]-p1[1])**2
    return np.sqrt(S)
```

10.
```
def test_bords2D(p:list, L:float, R:float)->bool:
    if p[0] < R or p[0] > L-R or p[1] < R or p[1] > L-R:
        return False
    else:
        return True
```

11.
```python
def test_nonchevauchement2D(p1:list, p2:list, R:float)->bool:
    if distance(p1,p2) < 2*R:
        return False
    else:
        return True
```

12.
```python
def placement2D(N:int, R:float, L:float)->list:
    Resultat = []
    while len(Resultat) < N:
        x = L*np.random.rand()
        y = L*np.random.rand()
        test = True
        for p in Resultat:
            if not test_nonchevauchement2D([x,y], p, R):
                test = False
        if test and test_bords2D([x,y],L,R):
            Resultat.append([x,y])
    return Resultat
```

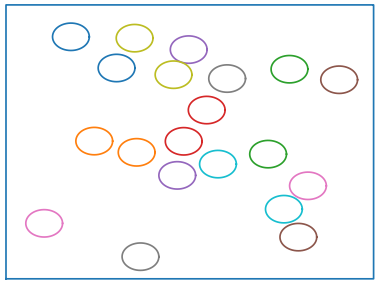
```
>>> L = 10
>>> R = 0.5
>>> N = 20
>>> Resultat = placement2D(N, R, L)
>>> Resultat
[[3.554069284347875, 4.6178418079323125], [7.1300746736472025, \
↪ 4.552111608000352], [4.834320274829422, 5.020112221090147], \
↪ [4.970126413435155, 8.368156771149916], [9.063478136664585, \
↪ 7.267575840290229], [8.214438635836297, \
↪ 3.3955813393036918], [3.6580196138477437, \
↪ 0.8084765839806884], [3.4986160912810513, \
↪ 8.786128679998734], [5.766238741569817, 4.187923292596074], \
↪ [1.765874055802007, 8.835844955370575], [2.400582784312401, \
↪ 5.027746535985375], [7.711532405439007, 7.652300715016731], \
↪ [5.459828734272851, 6.16114558163999], [4.6585749561319245, \
↪ 3.776507160325917], [7.952977445317528, \
↪ 1.5220133842269867], [1.0385572128002385, \
↪ 2.0251744358078994], [6.015725367473722, 7.31129239062887], \
↪ [4.557926735630917, 7.452529660600339], \
↪ [7.5595850675950675, 2.5422040201186116], \
↪ [3.0075673231299693, 7.696283788306677]]
```

Non demandé, mais on peut les tracer.

```python
plt.axis("off")
plt.plot([0, L, L, 0, 0], [0, 0, L, L, 0]) # le carré

def cercle(centre, R):
    X = np.linspace(0, 2*np.pi, 10**3)
    Y = np.cos(X)
    Z = np.sin(X)
    plt.plot(centre[0]+R*Y, centre[1]+R*Z)

for particule in Resultat:
    cercle([particule[0], particule[1]], R)
```



PARTIE IV — MOUVEMENT DES PARTICULES EN DIMENSION 1

13. 13.1) Si $m_1 = m_2$, on trouve $\overrightarrow{v_1'} = \overrightarrow{v_1}$ et $\overrightarrow{v_2'} = \overrightarrow{v_2}$, autrement dit, les particules échangent leur vitesse.

13.2) Si m_1 est très petit devant m_2, alors, on a approximativement $\overrightarrow{v_1'} = -\overrightarrow{v_1} + 2\overrightarrow{v_2}$ et $\overrightarrow{v_2'} = \overrightarrow{v_2}$. Il s'agit ici du cas d'un rebond sur la paroi : la vitesse de la particule est changée en son opposé.

14. 14.1) On a $\overrightarrow{\mathrm{M}(0)\mathrm{M}(t)} = t\vec{v}$, ce qui nous conduit à écrire :

```python
def vol(p:list, t:float) -> None:
    p[0] = p[0]+t*p[1]
```

14.2)
```python
def rebond(p:list) -> None:
    p[1] = -p[1]
```

14.3)
```python
def choc(p1:list,p2:list) -> None:
    p1[1], p2[1] = p2[1], p1[1]
```

PARTIE V — INVENTAIRE DES ÉVÈNEMENTS EN DIMENSION 1

15. 15.1)
```
def tr(p:list, R:float, L:float) -> float or None:
    if p[1] < 0:
        return (R-p[0])/p[1]
    elif p[1] > 0:
        return (L-R-p[0])/p[1]
    else:
        return None
```

15.2) On a $x_1(t) = x_1(0) + t\,v_1$ et $x_2(t) = x_2(0) + t\,v_2$, ce qui implique :

$$x_2(t) - x_1(t) = x_2(0) - x_1(0) + t(v_2 - v_1).$$

Les particules entreront en contact s'il existe un réel $t > 0$ tel que $x_2(t) - x_1(t) = 2\mathrm{R}$. On peut par ailleurs remarquer que, les particules ayant été disposées initialement sans chevauchement, on a nécessairement $|x_2(0) - x_1(0)| \geq 2\mathrm{R}$. Envisageons alors plusieurs cas :

- Si $v_2 - v_1 = 0$, les particules n'entreront jamais en contact.
- Si $v_2 - v_1 \neq 0$, on retourne la seule solution positive de l'équation $|x_2(0) - x_1(0) + t(v_2 - v_1)| = 2\mathrm{R}$. Or :

$$|x_2(0) - x_1(0) + t(v_2 - v_1)| = 2\mathrm{R} \iff t = \frac{\pm 2\mathrm{R} - x_2(0) + x_1(0)}{v_2 - v_1}.$$

Ce qui nous conduit à la fonction ci-dessous :

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

16. 16.1)
```
def ajoutEv(catalogue:[[float,int,int or None]], \
↪ e:[float,int,int or None]) -> None:
    i = 0
    while i < len(catalogue)  and catalogue[i][0] > e[0] \
    ↪ : # calcul de l'indice où il faut ajouter e
        i = i+1
    catalogue.insert(i,e)
```

16.2) Pour chaque particule de numéro différent de i, il y a au plus un évènement à ajouter (une collision). Il faut aussi ajouter une éventuelle collision avec la paroi pour la particule i.

```
def ajout1p(catalogue:[[float, int, int or None]], i:int, \
↪ R:float, L:float, particules:[[list]]) -> None:
    for j in range(len(particules)):
        if j != i:
            t = tc(particules[i],particules[j],R)
            if t != None:
                ajoutEv(catalogue, [t, i, j])
    t = tr(particules[i],R,L)
    if t != None:
        ajoutEv(catalogue, [t, i, None])
```

16.3) On part d'un catalogue vide et on l'enrichit en ajoutant les évènements à venir liés à chaque particule.

```
def initCat(particules:[[list]], R:float, L:float) -> \
↪ [[float, int, int or None]]:}
    N = len(particules)
    catalogue = []
    for i in range(N):
        ajout1p(catalogue, i , R, L, particules)
    return catalogue
```

16.4) 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 la rencontre des particules i et j dans le catalogue des événements.