DS d'informatique N°3

Quelques questions demandent d'écrire du code python, vous veillerez autant que possible à utiliser une syntaxe valide et à indenter correctement le code (utilisez des lignes verticales pour marquer les différents niveaux d'indentation). Un rappel sur les listes en python est donné à la fin de l'énoncé. Le sujet est probablement trop long pour être traité en 1 h

Détection de collisions entre particules

On considère un ensemble de n particules en mouvement dans un espace à deux dimensions, délimité par un rectangle de dimensions (non nulles) *largeur* × *hauteur*. L'objectif est de faire évoluer le système jusqu'à ce que deux particules entrent en collisions.

Tout au long de ce sujet, il est possible d'utiliser les fonctions demandées dans les questions précédentes du sujet, même si ces questions n'ont pas été traitées.

On rappelle que l'on peut récupérer directement les valeurs contenues dans un tuple (qui est une liste non modifiable) de la façon suivante : après l'instruction `a,b,c = (1,2,4)`, la variable `a` contient la valeur `1`, `b` contient la valeur `2` et `c` contient la valeur `4`. Cette instruction génère une erreur si le nombre de variables à gauche est différent de la taille du tuple à droite.

Un tableau `m` à deux dimensions $\ell \times c$ est un tableau de tableaux, ou plus précisément un tableau de longueur ℓ (nombre de lignes) contenant dans chaque case un tableau de longueur c (nombre de colonnes). La case `m[i][j]` correspond ainsi à l'élément qui se trouve sur la ligne d'indice `i`, dans la colonne d'indice `j`.

Partie I. Simulation du mouvement des particules

Comme indiqué plus haut, on considère un ensemble de n particules en mouvement dans un espace à deux dimensions, délimité par un rectangle de dimensions (non nulles) *largeur* × *hauteur*.

On considère que le temps est discret. La simulation commence au temps $t = 0$, et à chaque étape, on calcule la configuration au temps $t + 1$ en fonction de la configuration au temps t.

À tout instant t donné, chaque particule est définie par un quadruplet (x, y, vx, vy), où (x, y) sont ses coordonnées réelles représentées par des nombres flottants et où (vx, vy) est son vecteur vitesse, lui aussi constitué de deux nombres flottants.

Dans tout le sujet, on suppose que la norme de la vitesse de toute particule est majorée par une constante v_{max} .

Pour calculer les paramètres au temps $t + 1$ d'une particule qui, au temps t, est en position (x, y) avec un vecteur vitesse (vx, vy), on procède successivement aux traitements suivants (un exemple d'exécution est donné en fin de partie I) :

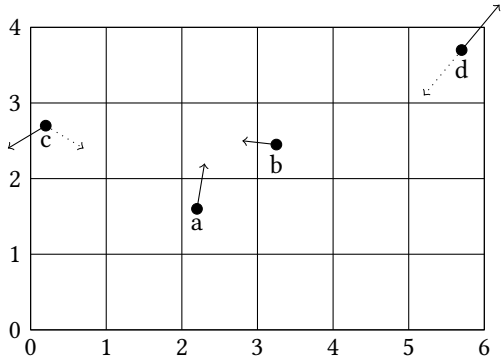
(i) si $x + vx$ atteint ou dépasse une paroi verticale, vx est changé en $-vx$ pour simuler le rebond ;
(ii) si $y + vy$ atteint ou dépasse une paroi horizontale, vy est changé en $-vy$ pour simuler le rebond ;
(iii) (x, y) est changé en $(x + vx, y + vy)$.

Les points (i) et (ii) simulent de façon simplifiée les rebonds sur les parois : on considère que la particule rebondit à l'endroit où elle est au temps t, ce qui nous permet d'éviter de calculer le véritable point de collision avec la paroi. Il y a rebond lorsqu'une particule arrive exactement sur la paroi ou qu'elle la dépasse. Il est possible qu'une particule rebondisse sur une paroi verticale et une horizontale pendant une même mise à jour, ce qui correspond au rebond sur un coin.

Important : Au départ, aucune particule n'est sur la paroi. On suppose de plus que

$$v_{max} < \frac{1}{2}\min(largeur, hauteur),$$

ce qui garantit que les particules restent toujours strictement à l'intérieur des parois.



Dans l'exemple ci-dessus, le rectangle est de dimension *largeur* $\times$ *hauteur* $= 6 \times 4$. Les particules **a** et **b** se déplacent sans rebondir au temps $t+1$. La particule **c** est sujette au point (i), comme $x + vx \leqslant 0$, elle rebondit sur la paroi, ce que l'on simule en changeant vx en $-vx$ avant d'effectuer le déplacement (le nouveau vecteur vitesse est représenté en pointillés). La particule **d** est sujette aux deux points (i) et (ii), puisque $x + vx \geqslant$ *largeur* et $y + vy \geqslant$ *hauteur*, on change donc vx en $-vx$ et vy en $-vy$ avant de déplacer cette particule.

Question 1. Écrire une fonction `deplacerParticule(particule, largeur, hauteur)` qui prend en paramètre une particule à l'instant t ainsi que les dimensions du rectangle et renvoie la particule à l'instant $t+1$, en tenant compte des rebonds.

On rappelle qu'en Python l'instruction `x,y,vx,vy = particule` permet de récupérer directement les différentes valeurs caractérisant la particule `particule`.

Exemples d'executions de cette fonction :

```
>>>deplacerParticule((2.2, 1.6, 0.1, 0.6), 6, 4)
(2.3, 2.2, 0.1, 0.6)
>>>deplacerParticule((3.25, 2.45, 0.45, 0.05), 6, 4)
(3.7, 2.5, 0.45, 0.05)
>>>deplacerParticule((0.2, 2.7, -0.5, -0.3), 6, 4)
(0.7, 2.4, 0.5, -0.3)
>>>deplacerParticule((5.7, 3.7, 0.5, 0.6), 6, 4)
(5.2, 3.1, -0.5, -0.6)
```

Partie II. Représentation par une grille

Dans un premier temps, on décide de représenter un ensemble de particules par une grille. Une particule de coordonnées (x, y) se trouvera obligatoirement dans la case d'indices $\lfloor x \rfloor$, $\lfloor y \rfloor$; la notation $\lfloor x \rfloor$ correspond à la partie entière inférieure de x, c'est à dire au plus grand entier inférieur ou égal à x. En Python, on obtient la partie entière inférieure d'un flottant `x` positif ou nul en utilisant la fonction `int(x)`.

Comme nous avons vu qu'une particule se trouve toujours à l'intérieur du rectangle et jamais sur les parois, cette simplification n'entraîne aucun débordement de tableau.

Une case de la grille ne peut contenir qu'une seule particule : si deux particules ou plus devaient aboutir dans la même case, on considère qu'il y a une collision et la simulation se termine.

Pour indiquer qu'une case est vide (sans particule), on utilisera `None`.

Attention : cette simplification où l'on considère que deux particules sont en collision lorsqu'elles sont dans la même case est utilisée uniquement dans cette partie.

En Python, cette grille sera représentée par un tableau à deux dimensions (le nombre de lignes correspond à la largeur et le nombre de colonnes à la hauteur de la grille). Il s'agit donc d'un tableau de tableaux, ou plus précisément d'un tableau de longueur *largeur* contenant dans chaque case une colonne, qui est elle-même un tableau de longueur *hauteur*. La case d'indices $[i][j]$ dans ce tableau correspondra ainsi à la case de cordonnées (i, j) dans la grille.

Voici le tableau à deux dimensions en Python correspondant à la grille donnée en exemple :

```
[[None,None,(0.2, 2.7, -0.5, -0.3), None],
 [None, None, None, None],
 [None, (2.2,1.6, 0.1, 0.6), None, None],
```

```
 [None, None, (3.25, 2.45, 0.45, 0.05) , None],
 [None, None, None, None],
 [None, None, None, (5.7, 3.7, 0.5, 0.6)] ]
```

Question 2. Écrire une fonction `nouvelleGrille(largeur, hauteur)` qui renvoie une nouvelle grille vide de dimensions `largeur` × `hauteur`.

Question 3. Pour cette partie, on considère qu'une collision entre deux particules survient si elles arrivent dans la même case de la grille à un instant donné. Écrire une fonction nommée `majGrilleOuCollision(grille)` qui prend en paramètre une grille contenant des particules à l'instant t et renvoie une nouvelle grille contenant ces particules à l'instant $t+1$ s'il n'y a pas eu de collision. Si une collision survient, la fonction renvoie `None`.

Remarque : Attention à ne pas confondre les particules à l'instant t avec celles à l'instant $t+1$.

Question 4. Écrire une fonction `attendreCollisionGrille(grille, tMax)` qui prend une grille de particules en paramètre et renvoie le temps où a eu lieu la première collision entre deux particules. S'il n'y a pas de collision avant le temps `tMax`, la fonction renvoie `None`.

Question 5. Quelle est la complexité de la fonction `attendreCollisionGrille(grille, tMax)` en fonction des dimensions (*largeur* et *hauteur*) de la grille et de `tMax` ? La réponse devra être justifiée.

Partie III. Représentation par une liste de particules

La représentation de l'ensemble des particules sous forme de grille est un peu contraignante du fait que l'on ne peut pas avoir deux particules dans la même case, ce qui nous a obligés à simplifier la notion de collision. On propose dans cette partie une représentation alternative, où l'on stocke les particules sous forme d'une liste, ce qui nous permettra de gérer plus finement les collisions.

Un **ensemble de particules** est représenté par un triplet `(largeur, hauteur, listeParticules)` tel que *largeur* × *hauteur* sont les dimensions du rectangle et `listeParticules` est la liste des particules considérées.

Avec cette nouvelle représentation, on considère également que les particules ont un rayon fixe et identique pour toutes les particules. La valeur de ce rayon est stockée dans une variable globale `rayon` (une variable globale est accessible en lecture n'importe où dans le code, même à l'intérieur des fonctions). Une collision entre deux particules survient lorsqu'elles se touchent, en prenant en compte le rayon.

Exemple : Voici un ensemble de particules correspondant à l'exemple donné en introduction.

```
(6, 4, [(2.2, 1.6, 0.1, 0.6),
        (3.25, 2.45, 0.45, 0.05),
        (0.2, 2.7, -0.5, -0.3),
        (5.7, 3.7, 0.5, 0.6)]
)
```

Listes non triées

Dans un premier temps, il n'y a aucune contrainte sur l'ordre des particules dans la liste.

Question 6. Écrire une fonction `detecterCollisionEntreParticules(p1, p2)` qui prend en paramètre deux particules et renvoie `True` si les particules sont en collision et `False` sinon.

Question 7. Écrire une fonction `maj(particules)` qui prend en paramètre un ensemble de particules (un triplet comme indiqué plus haut) à l'instant t et renvoie un ensemble contenant les particules à l'instant $t+1$, sans s'occuper des collisions éventuelles.

Question 8. À l'aide de la fonction précédente, écrire une fonction `majOuCollision(particules)` qui prend en paramètre un ensemble de particules (un triplet comme indiqué plus haut) à l'instant t et qui renvoie un ensemble contenant les particules à l'instant $t+1$, s'il n'y a pas eu de collision à l'instant $t+1$. S'il y a eu une collision, la fonction renvoie `None`.

Question 9. Écrire une fonction `attendreCollision(particules, tMax)` qui prend un ensemble de particules et un temps `tMax` en paramètres et renvoie le temps où a eu lieu la première collision entre deux particules. S'il n'y a pas de collision avant le temps `tMax`, la fonction renvoie `None`. Quelle est sa complexité, en fonction du nombre `n` de particules et de `tMax` ? La réponse devra être justifiée.

Listes triées

Afin d'essayer d'améliorer l'efficacité de la détection des collisions, on propose de trier la liste des particules selon leurs abscisses. L'idée est qu'une particule p ne peut entrer en collision qu'avec des particules suffisamment proches d'elle, et il ne sera donc pas nécessaire de parcourir toute la liste pour trouver les particules susceptibles d'entrer en collision avec p.

On rappelle que les normes des vitesses de toutes les particules sont majorées par v_{max} . On supposera que l'on dispose d'une variable globale `vMax` qui contient cette valeur.

Question 10. Pour que deux particules `a` et `b` aient une chance d'entrer en collision à un instant $t+1$ donné, à quelle distance, au maximum, devaient-elles se trouver à l'instant t ? On exprimera le résultat en fonction du rayon `rayon` des particules et de leur vitesse maximale `vMax`.

Question 11. Écrire la fonction `majOuCollisionX(particules)`. Elle prend en paramètre un ensemble de particules dont la liste des particules est triée par abscisses croissantes. Elle renvoie un ensemble contenant les particules à l'instant $t+1$, sauf si une collision survient entre deux particules, auquel cas la fonction renvoie `None`. Cette fonction devra exploiter le fait que la liste des particules est triée pour limiter le nombre d'appels à la fonction `detecterCollisionEntreParticules`.

Remarque : On ne demande pas que la liste de particules du résultat, s'il y en a une, soit triée.

Rappels sur les listes

Sur les listes, on dispose des opérations suivantes, qui ont toutes une complexité constante :

- `[]` crée une liste vide (c'est-à-dire ne contenant aucun élément)
- `len(liste)` renvoie la longueur de la liste `liste`.
- `liste[i]` renvoie l'élément d'indice `i` de la liste liste s'il existe ou produit une erreur sinon (noter que les éléments sont indicés à partir de `0`).
- `liste[-i]` renvoie l'élément d'indice `len(liste)-i` de la liste liste s'il existe ou produit une erreur sinon. En particulier, `liste[-1]` renvoie le dernier élément de la liste.
- `liste.append(x)` ajoute le contenu de `x` à la fin de la liste `liste` qui s'allonge ainsi d'un élément. Par exemple, après l'exécution de la suite d'instructions "`liste = []; liste.append(2); liste.append([1,3]);`", la variable `liste` a pour valeur la liste `[2,[1, 3]]`. Si ensuite on fait l'instruction `liste[1].append([7,5]);`, la variable `liste` a pour valeur la liste `[2, [1, 3, [7,5]]]`.
- `liste.pop()` renvoie la valeur du dernier élément de la liste `liste` et l'élimine de la liste. Ainsi, après l'exécution de la suite d'instructions "`listeA = [1,[2,3]]; listeB = listeA.pop(); c = listeB.pop();`", les trois variables `listeA`, `listeB` et `c` ont pour valeurs respectives `[1]`, `[2]` et `3`.