Détection de collisions entre particules (X PSI-PT 2016)

Durée : 2 heures

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 collision.

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

Complexité

La complexité, ou le temps d'exécution, d'un programme P (fonction ou procédure) est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc.) nécessaires à l'exécution de P. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que P a une complexité en $O(\phi(n,m))$ lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de P est inférieure ou égale à $A.\phi(n,m)$, pour tout $n \geqslant n_0$ et $m \geqslant m_0$.

Lorsqu'il est demandé de donner une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

Python - listes et tuples

Dans ce sujet, nous adopterons la syntaxe du langage Python. On rappelle qu'en Python, les listes sont des tableaux dynamiques à une dimension. Sur les listes, on dispose des opérations suivantes, qui ont toutes une complexité constante :

- `[]` crée une liste vide (c.-à-d. 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`.

Important : l'usage de toute autre fonction sur les listes telle que `liste.insert(i, x)`, `liste.remove(x)`, `liste.index(x)`, ou encore `liste.sort(x)` est rigoureusement interdit (ces fonctions devront être programmées explicitement si nécessaire).

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.

On rappelle également que l'on peut récupérer directement les valeurs contenues dans un tuple 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.

Dans la suite, nous distinguerons *fonctions* et *procédures* : les fonctions renvoient une valeur (un entier, une liste, ...) tandis que les procédures ne renvoient aucune valeur.

Nous attacherons la plus grande importance à la lisibilité du code produit par les candidats ; aussi, nous encourageons les candidats à introduire des procédures ou fonctions intermédiaires lorsque cela simplifie l'écriture.

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) $\mathit{largeur} \times \mathit{hauteur}$.

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, v_x, v_y), où (x, y) sont ses coordonnées réelles représentées par des nombres flottants et où (v_x, v_y) 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 (v_x, v_y), on procède successivement aux traitements suivants (un exemple d'exécution est donné en fin de partie I) :

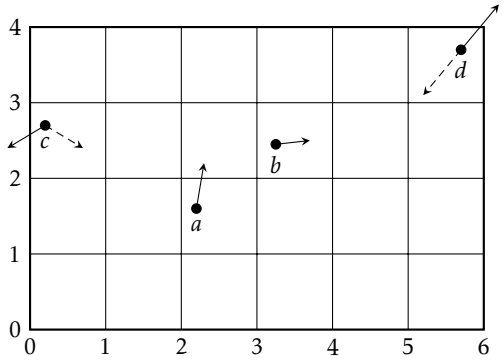
(i) si $x + v_x$ atteint ou dépasse une paroi verticale, v_x est changé en $-v_x$ pour simuler le rebond ;

(ii) si $y + v_y$ atteint ou dépasse une paroi horizontale, v_y est changé en $-v_y$ pour simuler le rebond ;

(iii) (x, y) est changé en $(x + v_x, y + v_y)$.

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 $v_{\max} < \frac{1}{2} \min(\mathit{largeur}, \mathit{hauteur})$, ce qui garantit que les particules restent toujours strictement à l'intérieur des parois.



Dans l'exemple ci-dessus, le rectangle est de dimension $\mathit{largeur} \times \mathit{hauteur} = 6 \times 4$. Les particules **a** et **b** se déplacent sans rebondir au temps $t + 1$. La particule **c** est sujette au point *(i)*, comme $x + v_x \leqslant 0$, elle rebondit sur la paroi, ce que l'on simule en changeant v_x en $-v_x$ 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 $x + v_x \geqslant \mathit{largeur}$ et $y + v_y \geqslant \mathit{hauteur}$, on change donc v_x en $-v_x$ et v_y en $-v_y$ 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'exécution 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 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 obligé 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.

À ce stade, on pourrait calculer le temps de première collision à partir d'appels à `majOuCollisionX(particules)` et de tris des listes de particules à chaque étape, en utilisant un tri efficace classique. On peut cependant remarquer que si les vitesses sont petites, l'ordre des particules au temps $t+1$ a peu changé par rapport au temps t. On souhaite donc plutôt utiliser un algorithme de tri qui tient compte de cette particularité. C'est l'objet de la partie suivante.

Partie IV. Trier des listes partiellement triées

On souhaite maintenant proposer un algorithme de tri, qui est d'autant plus efficace que la liste donnée en entrée est déjà partiellement triée. On ne donnera pas de définition formelle de ce que ce terme signifie. Dans toute cette partie, pour simplifier, *on ne triera que des listes d'entiers* (`int`).

Le tri choisi est une version simplifiée du tri utilisé par **Python** (qui s'appelle `TimSort`). On nommera α-tri cette version simplifiée. Ce tri est basé sur un découpage de la liste à trier en séquences croissantes maximales d'éléments consécutifs (appelées *scm*). Ces séquences sont croissantes au sens large. Il consiste à effectuer une succession de fusions de *scm* consécutives jusqu'à n'avoir plus qu'une seule *scm*. Fusionner deux *scm* consécutives consiste à réordonner leurs éléments pour ne former qu'une seule *scm*, comme dans le tri fusion. On notera $|x|$ la longueur d'une *scm* x.

On rappelle qu'une *pile* est une liste `pile` qui, outre son initialisation, possède deux opérations : l'ajout en fin de liste d'un élément `x` en utilisant `pile.append(x)`, et la suppression du dernier élément en utilisant `pile.pop()`. Cette dernière opération modifie la pile et renvoie l'élément supprimé, ou produit une erreur si `pile` est la liste vide.

L'algorithme α-tri se déroule en deux temps. On commence par partitionner la liste en *scm* consécutives, en identifiant leurs indices de début et de fin dans la liste. Dans un second temps, on effectue les fusions.

Partitionnement en *scm*

Si `s` est une liste d'entiers de longueur $n \geqslant 1$, son partitionnement en *scm* est l'unique séquence de longueur $k \geqslant 1$ de couples d'entiers (d_0, f_0) , (d_1, f_1), … (d_{k-1}, f_{k-1}) telle que :

- $d_0 = 0$ et $f_{k-1} = n-1$;
- $d_i \leqslant f_i$, pour tout $i \in \{0, \dots, k-1\}$;
- $d_{i+1} = f_i + 1$ pour tout $i \in \{0, \dots, k-2\}$;
- pour tout $i \in \{0, \dots, k-1\}$, la suite $\mathtt{s}[d_i], \mathtt{s}[d_i+1], \dots, \mathtt{s}[f_i]$ est croissante au sens large ;
- $\mathtt{s}[f_i] > \mathtt{s}[d_{i+1}]$ pour tout $i \in \{0, \dots, k-2\}$.

Exemple. Si l'on considère la séquence $\mathtt{s} = [3, 4, 8, 11, 1, 5, 2, 7, 9, 0, 10, 0]$, on obtient $k = 4$ et la décomposition :

$$\underbrace{3 \leqslant 4 \leqslant 8 \leqslant 11}_{(d_0,f_0)=(0,3)} \;>\; \underbrace{1 \leqslant 5}_{(d_1,f_1)=(4,5)} \;>\; \underbrace{2 \leqslant 7 \leqslant 9}_{(d_2,f_2)=(6,8)} \;>\; \underbrace{0 \leqslant 10}_{(d_3,f_3)=(9,10)} \;>\; \underbrace{0}_{(d_4,f_4)=(11,11)}$$

Question 12. Écrire une fonction `scm(s)` qui prend une liste `s` en paramètre et renvoie la liste ordonnée des couples d'indices correspondant au partitionnement en *scm* de `s`.

Par exemple, avec comme paramètre la liste `s = [3, 4, 8, 11, 1, 5, 2, 7, 9, 0, 10, 0]`, l'appel à `scm(s)` renverra la liste `[(0, 3), (4, 5), (6, 8), (9, 10), (11, 11)]`.

Fusions de deux *scm* consécutives

Les fusions effectuées par α-tri concernent toujours deux *scm* consécutives. Nous aurons donc besoin d'une procédure pour réaliser une telle fusion.

Question 13. Écrire une procédure `fusionner(s, r1, r2)` qui prend une liste `s` en paramètre ainsi que deux *scm* consécutives encodées par leurs indices de début et de fin, et les fusionne en une seule *scm* : si `r1` $= (d_1, f_1)$ et `r2` $= (d_2, f_2)$, alors après l'appel à la procédure, la partie de `s` située entre les indices d_1 et f_2 dans `s` doit être triée. Cette procédure ne crée pas une nouvelle liste, elle modifie la liste `s`.

Remarque. Il n'est pas demandé de vérifier que les *scm* sont consécutives. Si nécessaire, on supposera que l'on dispose d'une fonction `copier(s, debut, fin)` qui renvoie une copie de la liste donnée en paramètre entre les indices `debut` et `fin`, que l'on pourra utiliser pour recopier les sous-séquences de `s` qui correspondent aux *scm* décrites par `r1` et `r2`.

Algorithme α-tri

Les fusions des *scm* sont effectuées en deux temps. Tout d'abord, on utilise une pile initialement vide, dans laquelle les *scm* sont ajoutées une par une, dans l'ordre. À chaque fois qu'une *scm* est ajoutée, on compare les longueurs (leurs nombres d'éléments) de la dernière *scm* z de la pile et de l'avant-dernière y (si elle existe). Si $|y| < 2|z|$, on retire y et z de la pile, on les fusionne et on ajoute la *scm* fusionnée dans la pile. On continue à effectuer des fusions tant que la condition sur les longueurs des deux dernières *scm* est vérifiée. Quand on arrive à une pile avec un seul élément, ou telle que $|y| \geqslant 2|z|$, on ajoute la *scm* suivante dans la pile et on recommence les fusions éventuelles.

Dans un deuxième temps, lorsque toutes les *scm* initiales ont été ajoutées à la pile, on effectue une dernière passe en fusionnant itérativement les deux dernières *scm* de la pile, jusqu'à n'avoir plus qu'une seule *scm*. Cette *scm* est bien la liste initiale triée.

Exemple d'exécution de la phase de fusion pour `[3, 4, 8, 11, 1, 5, 2, 7, 9, 0, 10, 0]` :

```
** Découpage en scm **
Liste à trier: [3, 4, 8, 11, 1, 5, 2, 7, 9, 0, 10, 0]
Liste des scm: [(0, 3), (4, 5), (6, 8), (9, 10), (11, 11)]

** Première phase **
État de la pile: [(0, 3)]
État de la pile: [(0, 3), (4, 5)]
État de la pile: [(0, 3), (4, 5), (6, 8)]
Fusion des scm: (4, 5) et (6, 8). État de la liste: [3, 4, 8, 11, 1, 2, 5, 7, 9, 0, 10, 0]
État de la pile: [(0, 3), (4, 8)]
Fusion des scm}: (0, 3) et (4, 8). État de la liste: [1, 2, 3, 4, 5, 7, 8, 9, 11, 0, 10, 0]
État de la pile: [(0, 8)]
État de la pile: [(0, 8), (9, 10)]
État de la pile: [(0, 8), (9, 10), (11, 11)]

** Deuxième phase **
Fusion des scm: (9, 10) et (11, 11). État de la liste: [1, 2, 3, 4, 5, 7, 8, 9, 11, 0, 0, 10]
État de la pile: [(0, 8), (9, 11)]
Fusion des scm: (0, 8) et (9, 11). État de la liste: [0, 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
État de la pile: [(0, 11)]
La liste triée: [0, 0, 1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
```

Question 14. À l'aide de la procédure fusionner demandée à la question 13, écrire une procédure `depileFusionneRemplace(s, pile)` qui prend en paramètre une liste `s` ainsi qu'une pile de *scm* (sous la forme de couples d'indices de début et de fin). Cette procédure devra retirer les deux *scm* au sommet de la pile, les fusionner dans la liste `s` et replacer les indices de la *scm* fusionnée au sommet de la pile.

Remarque. La pile doit contenir au moins deux *scm* ; on suppose que c'est le cas, et il n'est donc pas demandé de le vérifier.

On rappelle que si `s` est une liste, `s[-1]` et `s[-2]` sont respectivement le dernier et l'avant-dernier élément de la liste, quand ils existent.

Question 15. En utilisant les questions précédentes, écrire une procédure `alphaTri(s)` qui prend en paramètre une liste `s` et trie cette liste en utilisant l'algorithme α-tri décrit ci-dessus (voir l'exemple). Attention, cette procédure ne crée pas une nouvelle liste, elle modifie la liste passée en paramètre.

L'algorithme `TimSort` *a d'abord été conçu pour le langage* Python. *Quelques années après, il a été adopté par d'autres langages de programmation. Il est notamment l'un des tris de la bibliothèque standard du langage* Java *depuis la version 7.*

♦ ♦

♦