Devoir Surveillé n° 2 : corrigé

MP Lycée Camille Jullian

17 décembre 2021

I. Initialisation.

I.A. Placement en dimension 1.

Question 1

La ligne 9 du programme tire au hasard un nombre flottant entre 0 et 1, puis le multiplie par L. Cela revient à choisir au hasard une position dans l'intervalle $[0, L[$, position qui va en l'occurrence être celle du centre de la particule qu'on essaye d'insérer.

Question 2

Le paramètre c représente simplement la position d'une particule, sous forme de tableau numpy (qui ne contient donc ici qu'un seul élément, ce qui est confirmé par la ligne 10 où on applique la fonction **possible** à la variable p qui est un tableau numpy à un seul élément.

Question 3

La ligne 3 permet de s'assurer que la position prévue pour le centre de la particule n'est pas trop près du bord du récipient : si $c[0] < R$, on est à une distance du bord gauche inférieure au rayon de la particule, et si $c[0] > L - R$, on est trop près du bord droit du récipient.

Question 4

On teste chacune des particules déjà présentes dans la liste pour voir si la position prévue pour le centre de la prochaine particule ne va pas créer un chevauchement : si la distance entre les centres de deux particules est inférieure à $2R$ (test effectué ligne 5), les deux particules vont en partie se retrouver au même endroit, ce qui n'est pas autorisé.

Question 5

La fonction **possible** sert à déterminer si une position de centre de particule à l'intérieur du récipient permet d'y ajouter une nouvelle particule compte tenu des particules déjà présentes dans le récipient.

Question 6

On aurait intérêt à ne pas positionner le centre de la particule dans tout l'intervalle $[0, L[$, mais plutôt dans l'intervalle $[R, L - R[$, pour ne pas avoir à tester à chaque fois la position par rapport aux bords. On échange donc la ligne 9 du programme contre la suivante :

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

Question 7

Les trois premières particules sont placées de telle façon qu'il est impossible de placer la quatrième et denrière particule prévue (chaque particule a un diamètre 1 et il reste quatre intevalles libres de largeur 0.5 chacun). La fonction **possible** va donc systématiquement renvoyer False, et la boucle while du programme ne s'arrêtera jamais puisque le test de la ligne 10 ne sera jamais positif.

Question 8

Avec la condition imposée sur N, on peut considérer que la fonction possible ne donnera jamais de rejet. Cette fonction a par ailleurs un coût en $O(k)$, où k est le nombre de particules déjà placées, puisqu'elle contient une boucle for avec autant d'étapes qu'il n'y a d'éléments présents dans la liste *res*. Par ailleurs, la boucle while présente dans le corps du programme va elle-même effectuer N fois de suite une opération à coût constant (le calcul de p) et un appel à la fonction **possible** qui sera de plus en plus coûteux au fur et à mesure du nombre de particules ajoutées. Si on compte le nombre global de tests effectués à la ligne 5 lors de l'exécution du programme, on en aura donc 1 pour l'ajout de la deuxième particule, puis 2 pour la troisième, ..., et enfin $N-1$ pour la dernière, soit au total $\frac{N(N-1)}{2}$ tests. Cela correspond à une complexité quadratique en $O(N^2)$.

Question 9

Il suffit de rajouter une option **else** derrière le test de la ligne 10, qui réinitialise la liste *res* si le test a été négatif :

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

I.B. Optimisation du placement en dimension 1.

Question 10

La première étape consiste simplement à calculer $l = L - 2RN$ (on enlève le diamètre $2R$ à l'espace disponible pour chacune des N particules), la deuxième va reprendre le principe du programme **placement1D** précédent, mais sans avoir recours à la fonction **possible** (ce qui simplifie les choses). Pour la troisième, on décale chaque centre de particule de R, et comme l'indique l'énoncé, on décale en même temps chacune des particules situées à droite de celle-ci de $2R$ (ce $2R$ correspondant au diamètre de la particule déplacée). Notons que cette façon de faire n'est pas optimale, il serait plus intelligent de trier la liste des positions des centres après l'étape 2 pour ensuite décaler le premier centre de R, le deuxième de $3R$ etc (on aurait une complexité sous-linéaire, alors qu'on va voir à la question suivante qu'ici on est en complexité quadratique).

```
def placement1Drapide(N,R,L) :
    l=L-2*R*N
    res=[]
    for i in range(N) :
        res.append(l*np.random.rand(1))
    for i in range(N) :
        p=res[i][0]
        for j in range(i-1) :
            if res[j][0]>p :
                res[j][0]+=2*R
        for j in range(i+1,N) :
            if res[j][0]>p :
                res[j][0]+=2*R
        res[i][0]+=R
    return res
```

Question 11

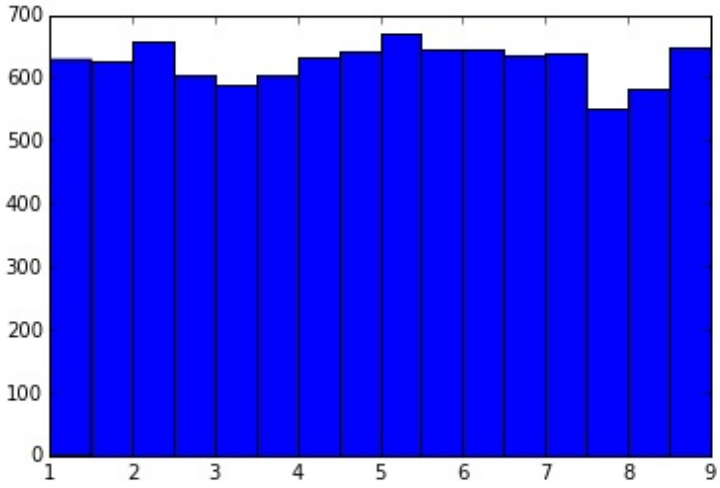
La première étape est en coût constant, la deuxième en $O(n)$ (en supposant comme toujours dans ce genre de cas que les append se font en coût constant). Par contre, la troisième étape nécessite une double boucle (la deuxième boucle est coupée en 2 comme j'ai rédigé le programme, mais la variable j prend $N-1$ valeurs au total), donc s'effectuera en temps quadratique $O(N^2)$.

I.C. Analyse statistique.

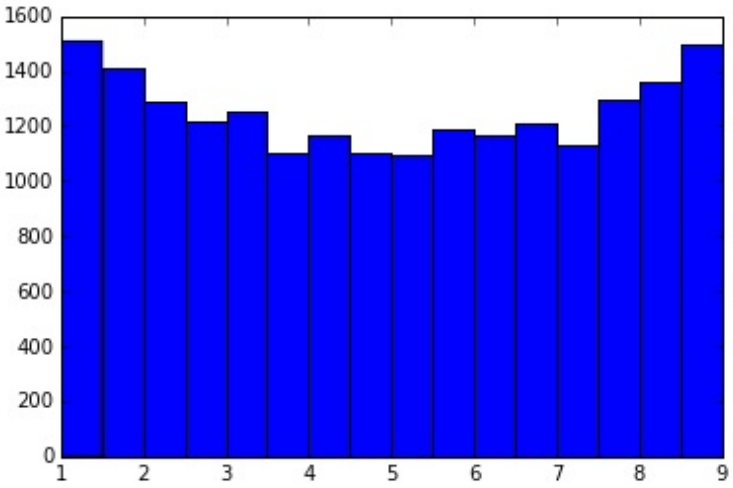
Question 12

Curieuse question où on ne sait pas trop ce qui est attendu ni comment exactement donner une idée de la représentation d'histogrammes pour lesquels la variable sur l'axe des abscisses peut prendre des valeurs continues au sein d'un intervalle. On va donc se contenter de pipoter :

- pour $N = 1$, on place une seule particule de diamètre 2 dans l'intervalle $[0, 10]$, son centre peut donc se trouver n'importe où dans l'intervalle $[1, 9]$. L'hiostogramme va donc donner un champ de barres uniformément réparties dans l'intervalle $[1, 9]$. Ci-dessous une simulation effectuée avec Python, en plaçant 10 000 particules (une par une, bien entendu), et en traçant un histogramme en regroupant les positions des centres par zones de largeur 0.5 (on remarquera que 10 000 particules ce n'est pas encore assez pour que la répartition soit réellement homogène) :



- pour $N = 2$, la même simulation pour 10 000 couples de particules donne la répartition suivante :



Peut-on l'expliquer, et notamment comprendre pourquoi les valeurs « au bord » semblent privilégiées par rapport à celles au centre ? C'est en fait un calcul mathématique assez compliqué. Pour simplifier les choses, séparons l'intervalle $[0, 6]$ où se trouvent les centres avant décalage (on suppose qu'on utilise le dernier algorithme programmé pour placer nos centres) en intervalles de largeur 0.5 comme on l'a fait pour tracer l'histogramme. La première particule a la même probabilité de se trouver dans chacun des 12 intervalles, et de même pour la deuxième. Après décalage, ces 288 particules virtuelles (on en a placé un couple dans chacun des 144 couples d'intervalles possibles) vont se répartir comme suit dans les seize sous-intervalles de $[1, 9]$ de largeur 0.5 : 23, 21, 19, 17, 16, 16, 16, 16, 16, 16, 16, 16, 17, 19, 21, 23 (par exemple, si on place notre couple dans les intervalles $[2, 2.5]$ et $[4.5, 5]$ initialement, elles se retrouveront après décalage dans les intervalles $[3, 3.5]$ et $[6.5, 7]$, je ne vois pas de façon évidente de calculer la répartition sans faire tous les cas, ce qui est évidemment impossible à représenter sur une copie, la seule façon raisonnable de faire semble donc être de restreindre le nombre d'intervalles de l'histogramme pour pouvoir tout faire à la main, mais dans ce cas on voit beaucoup moins clairement la prédominance des positions sur le bord). Ces valeurs correspondent assez bien à ce qu'on voit sur l'histogramme.

- pour $N = 5$, c'est beaucoup plus simple puisqu'on n'a qu'une seule disposition possible des cinq particules, toutes coincées les unes contre les autres avec des centres aux positions exactes 1, 3, 5, 7 et 9. Bien sûr, nos programmes n'arriveront jamais à recréer cette position parfaite, et l'histogramme n'aura donc aucun intérêt (des barres de même hauteur en 1, 3, 5, 7 et 9 et rien ailleurs).

I.D. Dimension quelconque.

Question 12

Il faut simplement adapter ce qu'on a fait en dimension 1 : pour que la position du centre d'une particule soit suffisamment éloignée de chacune des parois du récipient, il faut que chacune de ses cordonnées soit contenue dans l'intervalle $[R, L-2R]$, on se contentera donc d'appliquer à un tableau numpy de dimension 2 le même calcul que celui fait précédemment pour un tableau de dimension 1 (on rappelle en passant que les additions ou multiplications d'un tableau numpy par un flottant sont effectuées terme à terme). Pour vérifier la condition de distance entre deux centres de particules (qui intervient dans la fonction **possible**), il faut par contre effectuer un calcul de distance en dimension D, qu'on va donc programmer indépendemment (il prend comme arguments deux tableaux numpy, dont on calcule le nombre d'éléments).

```
def distance(u,v) :
    s=0
    for i in range(len(u)) :
        s+=(u[i]-v[i])**2
    return s**0.5

def placement(D,N,R,L) :
    def possible(c) :
        for p in res :
            if distance(p,c)<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
```

II. Mouvement des particules.

II.A. Analyse physique.

Question 14

Il est dit dans l'introduction que les particules ne sont soumises à aucune force dans le récipient, donc leur trajectoire en l'absence de choc sera rectiligne, à vitesse uniforme.

Question 15

Lorsque $m_1 = m_2$, les deux vitesses sont simplement échangées, ce qui est physiquement normal.

Question 16

On obtient $v_1' = 2v_2 - v_1$ et $v_2' = v_2$ (inutile d'utiliser des notations vectorielles ici, puisqu'on est en dimension 1). Ici, toutes les particules peuvent être considérées comme étant de masse similaire, le seul cas où on aura une situation de ce genre sera celui du rebond contre une paroi, avec en l'occurrence $v_2 = 0$ (les parois ne bougent pas), et donc $v_1' = -v_1$ (lors d'un rebond, la particule repart en direction opposée avec la même vitesse en valeur absolue).

II.B. Évolution des particules.

Question 17

Sans choc ni rebond, les coordonnées de la particule vont simplement être augmentées (ou diminutées) de $t \times v_i$, où $(v_1, \ldots, v_D)$ est le vecteur-vitesse de la particule. Autrement dit on calcule simplement (je ne mets rien derrière le return, il s'agit techniquement d'une procédure et pas d'une fonction) :

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

Question 18

Si on a bien compris l'énoncé, on voit qu'il suffit de changer la d-ème coordonnée du vecteur vitesse de la particule en son opposé.

```
def rebond(p,d) :
    p[1][d]=-p[1][d]
    return
```

Question 19

On échange donc tout simplement les vitesses des deux particules.

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

III. Inventaire des évènements.

III.A. Prochains évènements dans un espace à une dimension.

Question 20

Puisqu'on est à une dimension, la vitesse de la particule se réduit à un réel v, et le prochain rebond se fera sur la paroi de droite du récipient si $v > 0$, sur la paroi de gauche si $v < 0$ (et jamais si $v = 0$,

bien entendu). Si on note c la position initiale du centre de la particule, dans le cas où $v < 0$, le point le plus à gauche de la particule atteindra le bord gauche du récipient quand on aura $c - R + t \times v = 0$, soit $t = \dfrac{R - c}{v}$. De même, si $v > 0$, le point le plus à droite de la particule atteindra le bord droit du récipient quand on aura $c + R + t \times v = L$, soit $t = \dfrac{L - R - c}{v}$. Bien sûr, pusiqu'on est en dimension 1, le rebond se fait forcément perpendiculairement à la dimension 0 puisque c'est la seule disponible. D'où le superbe programme suivant :

```
def tr(p,R,L) :
    v=p[1][0]
    if v>0 : return ((L-R-p[0][0])/v,0)
    elif v<0 : return ((R-p[0][0])/v,0)
    return
```

Question 21

Les particules se rencontreront à l'instant t si leurs centres sont à une distance égale à $2R$, donc (en notant c_1 et c_2 les positions initiales des centres, et v_1 et v_2 les vitesses initiales des particules) si $c_1 + t \times v_1 = c_2 + t \times v_2 + 2R$ ou $c_1 + t \times v_1 = c_2 + t \times v_2 - 2R$. Autrement dit, on doit avoir $t = \dfrac{2R + c_2 - c_1}{v_1 - v_2}$ ou $t = \dfrac{2R + c_1 - c_2}{v_2 - v_1}$. Si $v_1 = v_2$, il n'y aura pas de solution (c'est normal, les particules ont la même vitesse initiale et resteront toujours à la même distance l'une de l'autre), dans le cas contraire on gardera la plus petite des deux valeurs obtenues si elles sont positives (les particules se rapprochent l'une de l'autre, la plus petite valeur de t correspond alors au moment où elles se percutent, et la plus grande à celui où elles auront fini de se traverser si on considère qu'elles continuent leur trajectoire comme si le choc n'avait pas eu lieu ; si les valeurs sont négatives c'est que les particules s'éloignent l'une de l'autre et ne se percuteront jamais).

```
def tc(p1,p2,R) :
    v1,v2,c1,c2=p1[1][0],p2[1][0],p1[0][0],p2[0][0]
    if v1==v2 : return
    t1,t2=(2*R+c1-c2)/(v2-v1),(2*R+c2-c1)/(v1-v2)
    if t1<0 : return
    if t1<t2 : return t1
    return t2
```

III.B. Catalogue d'évènements.

Question 22

Les courageux feront une dichotomie pour optimiser la complexité mais rien n'étant imposé, on peut se contenter de parcourir la liste jusqu'à atteindre un évènement qui se déroulera avant celui qu'on veut ajouter.

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

Question 23

On doit ajouter les collisions de la particule i avec chacune des autres, ainsi que sa collision avec une paroi. On va bien sûr reprendre les programmes écrits précédemment pour calculer le temps d'attente avant ces évènements, et les placer au bon endroit du catalogue.

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

Question 24

Il suffit tout simplement d'ajouter dans un catalogue initialement vide tous les évènements potentiels concernant chacune des particules disponibles.

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

Question 25

Comme on a programmé les choses, toutes les collisions entre particules apparaissent deux fois (une fois pour chaque particule), souci qu'on aurait pu aisément éviter en ne considérant que les chocs potentiels d'une particule avec les particules qui sont après elles dans la liste particules.

Question 26

En notant N le nombre de particules, on va appeler N fois la fonction **ajout1p**, qui va elle-même faire N appels à la fonction **ajoutEv**. Cette dernière a une complexité qui dépend de la taille du catalogue. La taille maximale du catalogue sera potentiellement de l'ordre de N^2 évènements ($N+1$ évènements pour chacune des N particules) et en moyenne en $O(N^2)$ également, ce qui fera une complexité globale en $O(N^4)$.

Question 27

On l'a en fait déjà signalé en la programmant, la fonction **ajoutEv** peut être largement améliorée en ayant recours à la dichotomie. On aurait alors un **ajoutEv** qui tournerait en $O(\ln(N))$ pour les plus gros catalogues à traiter, et une fonction **initCat** dont la complexité descendrait à du $O(N^2 \ln(n))$, ce qui est déjà beaucoup plus raisonnable.

IV. Simulation.

Question 28

S'il y a au moins une particule en mouvement initialement, ce sera toujours le cas (un rebond laissera la particule avec une vitesse non nulle, et un choc entre deux particules ne peut pas annuler simultanément leur deux vitesses), il y aura donc toujours des évènements ultérieurs.

Question 29

On va utiliser les programmes de la partie précédente : on doit faire « voler » les particules jusqu'à l'instant de l'évènement e, puis effectuer les modifications correspondant à cet évènement avec les fonctions **rebond** ou **choc** selon la nature de l'évènement.

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

Question 30

Il faut ici parcourir le catalogue pour modifier la date des futurs évènements, et rendre invalides tous ceux qui font intervenir les particules intervenant dans l'évènement e, puis ajouter au catalogue tous les nouveaux futurs évènements possibles à l'aide de la fonction **ajout1p** programmée précédemment.

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

Question 31

C'est en fait assez facile si on respecte bien tout ce qui est demandé : on crée une situation initiale aléatoire, et en même deux variables pour mesurer le temps écoulé depuis le début de la simulation et compter le nombre d'évènements qui se sont produit. Puis, tant que le temps n'a pas atteint la limite fixée, on effectue l'évènement valide suivant, on met tout à jour, et on enregistre l'évènement dans le base de données à condition qu'il se soit produit « à temps ».

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

Question 32

La deuxième occurrence du doublon sera déclarée non valide dès que la première aura été exécutée, ou pire, les deux seront déclarées non valides si un évènement faisant intervenir une deux particules concernées se produit auparavant.

Question 33

Le seul risque qu'on pourrait avoir est celui de l'imprécision des calculs de nouveaux temps du à la somme de temps très petits (écarts entre deux évènements) avec des valeurs plus grandes (temps écoulé depuis le début de la simulation) si l'ordre de grandeur des deux valeurs est très éloigné. Mais cela supposerait une simulation qui ferait intrvenir énormément d'évènements, et il est de toute façon probable dans ce cas que nos programmes de complexité non négligeable ne tournent pas efficacement quoi qu'il en soit. En termes de complexité, on perd nécessairement un peu de temps à réajuster les valeurs des temps d'attente des évènements, mais c'est négligeable par rapport à la complexité globale des fonctions programmées. Bref, pour conclure joliment ce superbe problème : il n'y a pas grand chose d'intéressant à répondre à cette question.