teaching sciences for innovation

Proposition de corrigé

Concours : Concours Centrale-Supélec

Année : 2018

Filière : MP - PC - PSI - TSI

Épreuve : Informatique

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'UPSTI (Union des Professeurs de Sciences et Techniques Industrielles), et publiée sur le site de l'association :

https://www.upsti.fr/espace-etudiants/annales-de-concours

A l'attention des étudiants

Ce document vous apportera des éléments de corrections pour le sujet traité, mais n'est ni un corrigé officiel du concours, ni un corrigé détaillé ou exhaustif de l'épreuve en question.

L'UPSTI ne répondra pas directement aux questions que peuvent soulever ces corrigés : nous vous invitons à vous rapprocher de vos enseignants si vous souhaitez des compléments d'information, et à vous adresser à eux pour nous faire remonter vos éventuelles remarques.

Licence et Copyright

Informez-vous !

Retrouvez plus d'information sur les Sciences de l'Ingénieur, l'orientation, les Grandes Ecoles ainsi que sur les Olympiades de Sciences de l'Ingénieur et sur les Sciences de l'Ingénieur au Féminin sur notre site : www.upsti.fr

L'équipe UPSTI

Correction IPT – CCS 2018
Simulation de la cinétique d'un gaz parfait

1 Initialisation

1.1 Placement en dimension 1

Q 1. La ligne 9 retourne un tableau contenant un flottant $p \in [0, L[$.

Q 2. Le paramètre `c` représente physiquement le centre de la sphère.

Q 3. Ce test a pour objectif de tester la position du centre de la sphère par rapport aux bords. En effet, pour que la sphère ne dépasse pas du récipient, il ne faut pas que son centre soit compris entre $[0, R[$ ou entre $]L - R, L]$.

Q 4. On teste ici, par rapport aux sphères peuplant déjà le récipient, afin d'éviter le chevauchement, la position du centre de la nouvelle sphère créée. Dès que cette distance est inférieure à $2 \times R$ alors le positionnement de la nouvelle sphère n'est pas possible.

Q 5. La fonction `possible` permet de vérifier si la nouvelle sphère s'intègre dans le récipient en vérifiant deux critères :

— elle ne dépasse pas du récipient ;

— elle ne se chevauche pas avec les autres sphères.

Q 6. Au lieu de tirer au sort entre $[0, L[$, il est possible de tirer au sort entre $[R, L - R[$. Pour cela, il suffit de remplacer la ligne 9 par : `p = R + (L-2*R)*np.random.rand(1)`. Par conséquent la ligne 3 devient inutile.

Q 7. Il est assez clair qu'il n'y a plus assez de place pour placer la quatrième sphère. Par conséquent la fonction `possible` renvoie en permanence `False`. Cette situation n'est pas gérée. On se retrouve alors dans une boucle infinie.

Q 8. L'hypothèse $N \ll N_{max}$ signifie qu'aucune sphère n'est rejetée par la fonction `possible`. Dans ce cas là, la ligne 8 est exécutée N fois. La complexité de la fonction `possible` est en $O(\text{len}(res))$ jusqu'à contenir N éléments. Par conséquent la complexité de l'ensemble est en $O(N^2)$.

Q 9. Il suffit de rajouter une commande `else` afin de redéfinir la liste `res` comme étant vide.

```
1  def placement1D(N:int, R:float, L:float) -> [np.ndarray]:
2      def possible(c:np.ndarray) -> bool:
3          if c[0] < R or c[0] > L - R : return False
4          for p in res :
5              if abs(c[0] - p[0]) < 2*R : return False
6          return True
7      res = []
8      while len(res) < N:
9          p = L * np.random.rand(1)
10         if possible(p) : res.append(p)
11         else : # ligne ajoutee
12             res = [] # ligne ajoutee
13     return(res)
```

1.2 Optimisation du placement en dimension 1

Q 10. Les deux premières étapes de l'algorithme ne pose pas de difficultés (lignes 2 et 3). Pour le troisième point, beaucoup de solutions peuvent être implémentées. Ici, je propose un tri après le

tirage au sort des N particules (ligne 4). Il suffit ensuite de décaler les particules suivantes du rayon de la particule courante et du diamètre de toutes les particules déjà placées (lignes 6 et 7). Il faut enfin faire attention au typage : cette fonction retourne une liste d'array. Ce qui donne le code suivant :

```
1 def placement1Drapide(N,R,L):
2     espace = L - N*2*R # point 1
3     positionsLibres = espace * np.random.rand(N) # tirage au sort des particules - point 2
4     positionsLibres.sort() # tri de la position des particules
5     positionsFinales = [] # liste des coordonnees des particules
6     for i in range(N):
7         positionsFinales.append(np.array([positionsLibres[i] + R + i*2*R]))
8     return(positionsFinales)
```

Q 11. La complexité de cette fonction dépend de la complexité du tri utilisé (forcément plus grande ou égale dans le meilleur cas que la complexité en $O(N)$ de la boucle `for`. La complexité moyenne d'un tri est en $O(N \log N)$. Par conséquent l'algorithme proposé ci-dessus a une complexité en $O(N \log N)$. Enfin, l'avantage réside (à mon sens dans) le déterminisme de la fonction et non dans sa complexité.

Commentaire : Le tri `sort` de Python s'appelle le *Timsort*. C'est un algorithme hybride du tri fusion et du tri par insertion. Par conséquent, sa complexité dans le pire cas est en $O(N \log N)$.

Commentaire : Si on ne pense pas à trier à la question précédente, on se retrouve avec un algorithme avec deux boucles `for` imbriquées (on décale les particules à droite). Alors cet algorithme serait en $O(N^2)$.

1.3 Analyse statistique

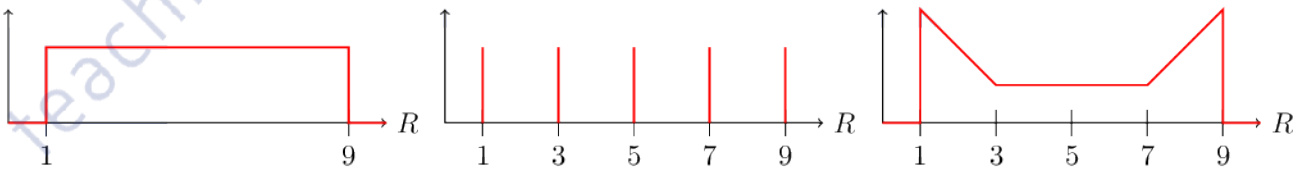
Q 12. Dans tous les cas, on doit avoir 0 sur $[0,1[$ et $]9,10]$ et une répartition symétrique par rapport à $x = 5$.

Pour $N = 1$: la position de la particule est équiprobable sur $[1,9[$, on doit obtenir une droite horizontale (Fig. 1a).

Pour $N = 5$: il n'y a qu'une seule possibilité pour placer toutes les particules. On obtient un segment vertical en 1, 3, 5, 7 et 9 (équiprobable) et rien ailleurs (Fig. 1b).

Pour $N = 2$, le décompte des différentes possibilités est plus compliqué. En notant x l'abscisse de la première particule et en dénombrant le nombre de possibilités restantes pour placer la deuxième, on obtient les résultat suivants :

- si $x \in [1,3[$, il n'est pas possible de placer la deuxième particule à sa gauche, il ne reste donc que le segment $[x+2,9[$ soit un espace de $7-x$;
- si $x \in [3,5]$, les emplacements possibles pour la deuxième particule sont $[1,x-2]$ et $[x+2,9[$, soit un espace de 4. On « doit » donc observer un maximum pour $x = 1$ puis une décroissance linéaire jusqu'à $x = 3$, un plateau entre 3 et 5 et une deuxième moitié de courbe symétrique par rapport à $x = 5$ (Fig. 1c).



(a) $N = 1$ (b) $N = 5$ (c) $N = 2$

Commentaire : Le mot « histogramme » n'est clairement pas le plus adapté.

1.4 Dimension quelconque

Q 13. On cherche ici à généraliser la fonction 1D. La difficulté induite est le calcul de la distance euclidienne entre deux centres de particules (lignes 5 et 6) et la position proposée de la particule en D dimensions (ligne 12 - `np.random.rand(D)`). Il ne faut pas oublier de prendre en compte les

améliorations des questions 6 et 9 (simplification de `possible` par le calcul de `p` (ligne 12). Cela donne :

```
1  def placement(D,N,R,L):
2      def possible(c):
3          for p in res:
4              distance = 0 # Initialisation de la distance
5              for i in range(D): # Calcul de la distance euclidienne
6                  distance + = (c[i]—p[i])**2
7              if np.sqrt(distance) < 2*R:
8                  return(False)
9          return(True)
10     res = []
11     while len(res) < N:
12         p = R + (L—2*R) * np.random.rand(D) # D dimensions — amelioration question 6
13         if possible(p):
14             res.append(p)
15         else:  # amelioration question  9
16             res = []
17     return(res)
```

Q 14. La particule possède un mouvement rectiligne uniforme entre deux événements selon les hypothèses (aucune action extérieure).

Q 15. Le calcul donne avec $m_1 = m_2 = m$:

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

Les deux particules de même masse « échangent » leur vitesse lors du choc (élastique).

Q 16. Cette fois-ci si $m_1 \ll m_2$, on trouve :

$$\vec{v_1}' \simeq -\vec{v_1} + 2\vec{v_2} \quad \text{et} \quad \vec{v_2}' \simeq 2\frac{m_1}{m_2}\vec{v_1} + \vec{v_2}$$

Cela correspond au choc contre une paroi. En effet celle-ci sera supposée de masse très grande devant celle des particules. Dans ce cas la vitesse $\vec{v_2}$ est également nulle, par conséquent :

$$\vec{v_1}' = -\vec{v_1} \quad \text{et} \quad \vec{v_2}' = \vec{v_2} = \vec{0}$$

La particule rebondit donc sans perte de vitesse (choc toujours élastique).

Q 17. Comme la vitesse est conservée (question 14) entre deux événements, il suffit, afin d'obtenir la nouvelle position, d'ajouter à l'ancienne le produit vitesse - temps. De plus il est possible d'utiliser les possibilités de calcul numérique liées au tableau en écrivant :

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

Commentaire : Si on ne pense pas à l'intérêt des tableaux, une boucle `for` sur la dimension du problème fait l'affaire.

Q 18. Pas de difficulté.

```
1  def rebond(p,d):
2      p[1][d] = —p[1][d]
```

Q 19. Cette fonction se résume à un échange des vitesses (qui plus est rappelé en annexe du sujet).

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

2 Inventaire des événements

2.1 Prochains événements dans un espace à une dimension

Q 20. Il s'agit de différencier trois cas : vitesse nulle où l'on renvoie `None` (lignes 2 et 3), si la vitesse est positive, on ira frapper la paroi de droite (lignes 5 et 6) et si elle est négative la paroi de gauche (7 et 8). La distance restant à parcourir est à droite $L - R - p[0][0]$ et à gauche $p[0][0] - R$. Ces distances sont à diviser par la vitesse pour obtenir le temps avant impact. Enfin, il ne faut pas oublier de renvoyer un tuple `(temps,dir)` avec `dir` la normale à la paroi (ici toujours 0 car en 1D).

```
1 def tr(p,R,L):
2     if p[1][0] == 0: # si vitesse nulle
3         return(None) # renvoie None
4     else: # sinon
5         if p[1][0] > 0:  # si positive
6             temps = (L-R-p[0][0])/p[1][0] # distance/ temps - ne pas oublier R
7         else: # si negative
8             temps = - (p[0][0]-R)/p[1][0] # distance / temps ((vitesse negative)
9         return(temps,0)
```

Commentaire : Il faut bien préciser le deuxième indice `[0]` pour travailler avec des flottants. Si on l'oublie, on obtient alors un tableau, ce qui pose problème dans la suite.

Q 21. Pour cette question, il faut différencier les différentes situations. Une idée est d'évaluer les différences de position `dx` (ligne 2) et de vitesse `dv` (ligne 3). Si le produit de ces deux grandeurs (ligne 4) est positif, il n'y aura jamais de choc ; sinon il suffit d'estimer le temps de choc (ne pas oublier que les particules ont un rayon R d'où le `-2*R`).

```
1 def tc(p1,p2,R):
2     dx = p2[0][0]-p1[0][0] # difference de position
3     dv = p2[1][0]-p1[1][0] # difference de vitesse
4     if dx*dv >= 0: # si produit positif, les particules s'ecartent
5         return(None) # None
6     return((abs(dx)-2*R)/abs(dv)) # temps de choc
```

2.2 Catalogue d'événements

Q 22. Il s'agit d'insérer dans une liste déjà triée l'événement à sa bonne place. Il faut d'abord géré le cas où la liste est vide (lignes 2 et 10). Si elle est vide on utilisera la méthode `append`. Sinon la liste étant ordonnée de manière décroissante, il est nécessaire que le temps associé à l'événement soit inférieur au temps lu, élément par élément, dans le catalogue (tout en vérifiant que l'on ne dépasse pas le dernier élément - ligne 4 - attention à l'ordre de la condition). À partir de l'indice `i` déterminé, on insère ce nouvel événement à sa place (ligne 7).

```
1 def ajoutEv(catalogue,e):
2     if catalogue : # test si catalogue non vide
3         i=0 # initialisation de l'indice
4         while  i < len(catalogue) and e[1] < catalogue[i][1] : # condition de position
5         # ordre obligatoire sinon out of range
6             i+=1 # increment
7         catalogue.insert(i,e) # insertion a la bonne place
8     else:
9         catalogue.append(e)  # si vide, on utilise append
```

Commentaire : Pour éviter de s'embêter dans l'ordre d'évaluation de la condition, une boucle `for` et un test `if` feraient l'affaire.

Q 23. Il faut regarder tous les événements pouvant avoir lieu. Pour cela, il ne faut pas oublier que les fonctions `tr` et `tc` renvoie `None` si celui-ci n'a pas lieu, d'où les lignes 3 et 8. Dans le code suivant,

on teste d'abord les rebonds (ligne 2), si celui-ci a lieu, on l'ajoute au catalogue (ligne 4). On réalise de même pour les chocs en balayant toutes les particules (ligne 5) en éliminant le test avec la même particule (ligne 6). Enfin si le choc a lieu (ligne 8), on l'ajoute au catalogue (ligne 9).

```
1 def ajout1p(catalogue,i,R,L,particules):
2     t = tr( particules[i],R,L) # Rebond ?
3     if t: # oui
4         ajoutEv(catalogue,[True,t[0],i,None,t[1]]) # ajout au catalogue du rebond
5     for j in range(len(particules)): # On balaie toutes les particules pour le choc eventuel
6         if i != j: # sauf la particule avec elle-meme !
7             t = tc(particules[i],particules[j],R) # Choc ?
8             if t: # Oui
9                 ajoutEv(catalogue,[True,t,i,j,None]) # ajout au catalogue le choc
```

Q 24. Il suffit de boucler sur l'ensemble des particules avec l'appel à `ajout1p`. La liste reste ordonnée grâce à l'appel de `ajoutEv` dans la fonction `ajout1p`.

```
1 def initCat(particules,R,L):
2     catalogue = [] # Initialisation
3     for i in range(len(particules)): # on balaie l'ensemble des particules
4         ajout1p(catalogue,i,R,L,particules) # on ajoute les evenements
5     return(catalogue)
```

Q 25. Le choc entre la i^e particule et la j^e particule est en double avec celui de la j^e et de la i^e.

Q 26. En appelant N le nombre de particules, la fonction `initCat` appelle N fois la fonction `ajout1p`. Cette dernière, entre autres, appelle N fois `ajoutEv`. Il reste donc à déterminer la complexité de cette dernière. Appelons N_e le nombre d'événements. La fonction `ajoutEv` a une complexité qui est au pire en $O(N_e)$, celle de `insert`. Il reste à relier le nombres d'événements au nombre de particules. Pour cela, chaque particule génère au pire N événements (elle rencontre toutes les autres plus une paroi) ; ce qui donne un majorant du nombre d'événements en N^2. Alors cette fonction `initCat` a une complexité en $O(N^4)$.

Commentaire : Pour être plus précis, en dimension 1, chaque particule va générer $\frac{N}{2}$ évènements en moyenne (elle rencontre une paroi et toutes les particules situées devant elle). Nous aurons donc $\frac{N^2}{2}$ évènements, dont N^2 est bien un majorant.

Q 27. On peut imaginer une fonction `ajout1p` construisant une nouvelle liste d'événements (indépendante du catalogue) et celle-ci serait triée avec un algorithme de tri en $O(N \log N)$. Puis en utilisant un tri fusion sur ces deux listes (complexité linéaire quand les deux sous-listes sont déjà triées), on peut se ramener à une complexité $O((N_e + N) \times \log(N)) = O(N^3 \log N)$.

Commentaire : Ce n'est qu'une solution parmi d'autres... Il en existe peut être des meilleures.

3 Simulation

Q 28. Dès qu'une particule est en mouvement, elle rencontrera au pire une paroi. Par conséquent, la seule solution pour que le catalogue d'événements soit vide est qu'aucune particule soit en mouvement.

Q 29. Il s'agit de mettre à jour les positions et vitesses des particules entre deux événements. La première chose à réaliser est d'extraire le temps relatif entre l'instant donné et l'événement considéré (ligne 2). Ensuite pour l'ensemble des particules, on met à jour leur position (lignes 3 et 4). Enfin, si l'événement considéré est valide (ligne 6), et en présence d'un choc (lignes 7 et 8), on met à jour les vitesses des particules i et j. Si l'événement est un rebond, on met à jour la vitesse de la particule i (lignes 9 et 10).

```
1 def etape(particules,e):
2     t=e[1] # extraction du temps de vol
3     for p in particules: # pour toutes les particules
4         vol(p,t) # nouvelle position
```

```
5     valide,i,j,paroi = e[0],e[2],e[3],e[4] # deconstruction pour aisance de lecture
6     if valide: # validite evenement
7         if j is not None: # c'est un choc
8             choc(particules[i],particules[j]) # mise a jour vitesse
9         if paroi is not None: # c'est un rebond  — un else suffit
10            rebond(particules[i],paroi) # mise a jour vitesse
```

Q 30. Cette fonction est à décomposer en trois parties : la première concerne la mise à jour des dates (lignes 2 à 4), la seconde l'invalidation des événements liés au choc ou rebond précédent (lignes 5 à 11) et la dernière sur l'insertion des nouveaux événements (lignes 12 à 16). La mise à jour se réalise en soustrayant le temps de l'événement (ligne 4). Pour l'invalidation, il faut balayer l'ensemble du catalogue (ligne 7), et si on rencontre un événement lié à une particule i ou j alors on rend l'événement invalide (lignes 9 et 11). Enfin il s'agit d'ajouter les différents éléments potentiels à l'aide de la fonction `ajout1p` (lignes 14 et 16).

```
1  def majCat(catalogue,particules,e,R,L):
2      # Mise a jour des dates
3      for k in range(len(catalogue)):
4          catalogue[k][1] -= e[1]
5      # Invalidation des evenements lies au choc ou rebond
6      i, j = e[2] , e[3] # particules concernees
7      for k in range(len(catalogue)): # on balie l'ensemble des evenements
8          if i is not None and catalogue[k][2] == i: # pour la particule i
9              catalogue[k][0] = False # on rend invalide ces evenements
10         if j is not None and catalogue[k][2] == j: # pour la particule j
11             catalogue[k][0] = False # on rend invalide ces evenements
12     # Insertion des evenements des particules concernees
13     if i is not None: # pour i
14         ajout1p(catalogue,i,R,L,particules) # ajout evenements
15     if j is not None: # pour j
16         ajout1p(catalogue,j,R,L,particules) # ajout evenements
```

Q 31. Il s'agit ici de mettre en musique l'ensemble des fonctions « élémentaires » du sujet afin de lancer la simulation. On distingue 2 parties : l'initialisation (lignes 2 à 5) de la boucle `while` sur la simulation (lignes 6 à 14). La phase d'initialisation concerne les particules (ligne 3), le catalogue d'événements (ligne 4) et les variables temps et nombre d'événements (ligne 5). Pour la simulation, il faut penser à extraire l'événement en dernière position (ligne 9) et si celui-ci est valide, on réalise les actions associées : `etape` puis `majCat` (lignes 10 et 11). On met à jour le temps et le nombre d'événement (ligne 12) puis on enregistre dans la base de données (ligne 13). Enfin on renvoie le nombre d'événements (ligne 14).

```
1  def simulation(bdd,D,N,R,L,T):
2      # Initialisation
3      particules = situationInitiale(D,N,R,L) # des particules
4      catalogue = initCat(particules,R,L) # du catalogue d'evenements
5      t,Ne = 0.0,0 # compteur temps, evenement
6      # boucle sur la duree de la simulation
7      while t <= T:
8          e = catalogue.pop() # extraction de l'evenement
9          if e[0]: # si celui-ci est valide
10             etape(particules,e) # etape
11             majCat(catalogue,particules,e,R,L) # mise a jour catalogue
12             t,Ne = t+e[1] , Ne+1 # mise a jour temps et evenements
13             enregistrer(bdd,t,e,particules) # mise a jour base de donnees
14     return(Ne)
```

Commentaire : Au fil du sujet le D de dimension s'est transformé en d. Un script adapté pour une animation graphique est proposé en marge de ce corrigé.

Q 32. En fait, dès que le premier événement est géré, le second - identique - est rendu invalide par la fonction `majCat`.

Q 33. L'avantage d'un temps absolu est qu'il n'est pas nécessaire de remettre à jour les différents temps. Par contre, il aurait fallu une « horloge » pour vérifier les instants. L'inconvénient majeur réside sans doute dans le souci de précision si le temps devient très grand ...

Commentaire : Très difficile de proposer quelque chose ici. J'ai l'impression que les deux configurations ont leurs avantages et inconvénients.

4 Exploitation des résultats

Q 34. Il s'agit ici de chercher dans la table `SIMULATION` l'ensemble des simulations suivant un critère de dimensions. Pour cela il est pratique de compter (`count(*)`) après regroupement selon `SI_DIM`.

```
SELECT SI_DIM, count(*) FROM SIMULATION GROUP BY SI_DIM ;
```

Q 35. Comme chaque rebond est enregistré, il faut les compter après les avoir regroupé pour chaque simulation. On projette suivant `SI_NUM` pour afficher le numéro de simulation ; on compte avec `count(*)` et on réalise la moyenne avec `AVG(RE_VIT)`.

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

Q 36. Il faut penser à sommer les variations à chaque choc (`SUM(2*PA_M*RE_VP)`). Ensuite il faut nécessairement une double jointure afin de regrouper les tables. De plus, il faut sélectionner selon le numéro de simulation (`WHERE`). Enfin, il faut regrouper selon la direction de la paroi (`GROUP BY`). Ce qui donne :

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