Lycée Faidherbe, 2020-21

Devoirs d'informatique

MP2&MP3

Version du 15 mars 2021

Table des matières

DS I

Coefficients binomiaux (MP)

Le but de ce problème est de définir les coefficients binomiaux et certaines de leurs utilisations.

Remarques

- Lorsque le résultat de la division de deux entiers doit être un entier on utilisera la division euclidienne : `n // p`.
- Lorsqu'un nombre d'opérations est demandé, il est possible de répondre sous la forme d'un grand $\mathcal{O}$.

1 Calculs des coefficients binomiaux

1.1 Calcul d'un coefficient

Exercice 1
Écrire une fonction `facto(n)` *qui calcule la factorielle de* n.
Combien de multiplications l'appel de `facto(n)` *effectue-t-il ?*

Exercice 2
En utilisant la définition $\binom{n}{p} = \dfrac{n!}{p!(n-p)!}$ *écrire une fonction* `binom_f(n, p)` *qui calcule* $\binom{n}{p}$.
Combien de multiplications et divisions l'appel de `binom_f(n, p)` *effectue-t-il ?*

Exercice 3
Exprimer $\binom{n}{k+1}$ *en fonction de* $\binom{n}{k}$ *pour* $0 \leqslant k < n$.

En déduire une fonction `binom_p(n, p)` *qui calcule* $\binom{n}{p}$ *en effectuant* p *multiplications et* p *divisions* **entières**.

Exercice 4
Pourquoi est-il indispensable d'utiliser la division euclidienne dans `binom_f(n, p)` *?*

1.2 Calcul des coefficients pour n fixé

On aura besoin de tous les coefficients binomiaux de la forme $\binom{n}{k}$ pour un n fixé ; on notera qu'il y en a $n+1$.
Par exemple `liste_binom(8)` doit renvoyer `[1, 8, 28, 56, 70, 56, 28, 8, 1]`.

Exercice 5
En utilisant la fonction `binom_f`*, écrire une fonction* `liste_binom1(n)` *qui renvoie la liste des coefficient binomiaux. On ne cherchera pas à optimiser les calculs, c'est l'objet de la question suivante.*
Combien de multiplications et de divisions sont effectuées lors de l'appel de `liste_binom1(n)` *?*

En fait on fait plusieurs fois les mêmes calculs de factorielles dans la fonction ci-dessus.

Exercice 6
Écrire une fonction `liste_binom2(n)` *qui renvoie la liste des coefficient binomiaux en faisant un nombre de produits et de divisions entières qui soit un $\mathcal{O}(n)$.*
On pourra commencer par créer une liste des $p!$ pour $0 \leqslant p \leqslant n$ en ne faisant que n multiplications.

Exercice 7
En utilisant la formule de l'exercice 3 mais sans utiliser `binom_p`*, écrire une fonction* `liste_binom3(n)` *de complexité en $\mathcal{O}(n)$ qui calcule la liste des coefficient binomiaux.*

1.3 Triangle de Pascal

On va ici calculer tous les coefficients binomiaux de la forme $\binom{q}{p}$ avec $0 \leqslant p \leqslant q \leqslant n$ pour un n fixé.

On peut utiliser les fonctions précédentes : on crée une liste de listes vides que l'on remplit avec les résultats de `liste_binom`

```
def pascal1(n):
    triangle = [[]]*(n+1)
    for q in range(n+1):
        triangle[q] = liste_binom3(q)
    return triangle
```

Par exemple `pascal1(4)` renvoie `[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1]]`.
Si `liste_binom3(q)` effectue q multiplications et q divisions alors `pascal1(n)` demande $\frac{n(n+1)}{2}$ multiplications et autant de divisions.
On peut remplacer les multiplications et divisions en utilisant la formule

$$\binom{q}{p} = \binom{q-1}{p-1} + \binom{q-1}{p} \text{ pour } 1 \leqslant p < q \leqslant n$$

Exercice 8
Écrire une fonction `pascal(n)` *qui renvoie le triangle de pascal comme ci-dessus en n'effectuant que des additions. Combien d'additions sont nécessaires pour calculer* `pascal(n)` *?*

Exercice 9
Comment utiliser `pascal(n)` *pour calculer la liste des $\binom{n}{p}$, $0 \leqslant p \leqslant n$, n fixé ?*

Cette méthode a un temps de calcul plus élevé mais peut être préférée car elle n'utilise que des additions.
Cependant elle demande beaucoup de place en mémoire ($\frac{(n+2)(n+1)}{2}$ valeurs).

Exercice 10 **
Écrire une fonction `liste_binom4(n)` *qui calcule la liste des $\binom{n}{p}$ pour $0 \leqslant p \leqslant n$ en n'effectuant que des additions et en ne définissant qu'une seule liste de taille $n+1$.*
Le nombre d'additions est inchangé par rapport à l'exercice 8.

2 Application

2.1 Polynômes de Bernstein

Pour une fonction continue sur $[0;1]$ on définit la suite des polynômes de Bernstein $\big(B_n(f)\big)$ par

$$B_n(f)(x) = \sum_{k=0}^{n} \binom{n}{k} f\left(\tfrac{k}{n}\right) x^k (1-x)^{n-k}$$

On peut démontrer que la suite $\big(B_n(f)\big)$ converge uniformément vers f sur $[a;b]$.
On voit qu'il suffit de connaître les valeurs de f aux $n+1$ points $\frac{k}{n}$ avec $0 \leqslant k \leqslant n$, pour définir $B_n(f)$ sur tout l'intervalle : les polynômes de Bernstein sont utilisés pour tracé des courbes définies par $n+1$ points, les courbes de Bézier d'ordre n.
On suppose donnée une fonction

```
def f(x):
    ....
    return y
```

Exercice 11
Écrire une fonction `bernstein(n, f, x)` *qui renvoie la valeur de* $B_n(f)(x)$.

2.2 Lissage

Lors d'une expérimentation les données sont acquises automatiquement à des intervalles de temps constant et sont restituées sous la forme d'une liste `x_exp` de taille N.
Malheureusement les incertitudes dans la mesure sont inévitables et les valeurs calculées ont une part d'aléatoire.
On va essayer de lisser la suite des valeurs en remplaçant chacune par une moyenne pondérée des valeurs voisines. On choisit un lissage de Gauss à l'ordre p :

$$\widehat{x}_i = \begin{cases} \dfrac{1}{2^{2p}} \displaystyle\sum_{j=-p}^{p} \binom{2p}{j+p} x_{i+j} & \text{pour } p \leqslant i \leqslant N-1-p \\ x_i & \text{pour } i < p \text{ ou } i \geqslant N-p \end{cases}$$

Par exemple, pour $N=4$ et $p=1$, $[x_0, x_1, x_2, x_3]$ devient $\left[x_0, \frac{x_0+2x_1+x_2}{4}, \frac{x_1+2x_2+x_3}{4}, x_3\right]$

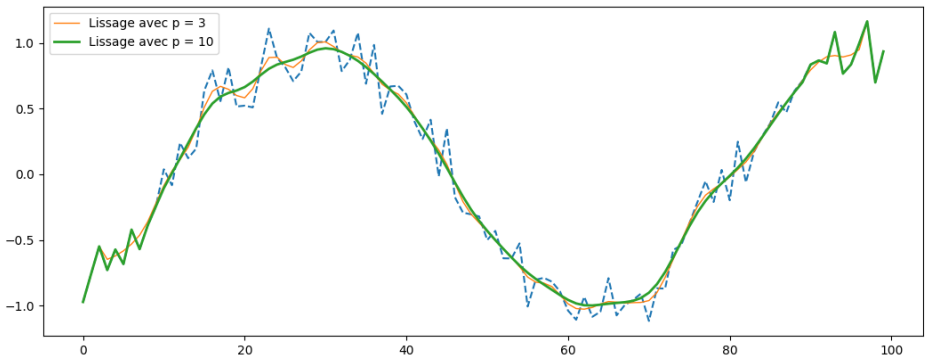


Figure I.1 – Exemples de lissage, les valeurs aux bordx ne sont pas lissées

Exercice 12
Pourquoi divise-t-on par 2^{2p} ?

Exercice 13
On a fait le choix de ne pas modifier les p premiers et les p derniers termes : proposer une autre possibilité. Cette question n'a pas de réponse unique.

Exercice 14
Écrire une fonction `lissage(x, p)` *qui renvoie la nouvelle liste des valeurs lissées obtenues à partie de la liste* `x` *en appliquant le lissage de Gauss à l'ordre p.*

Exercice 15
Quelle est la complexité de cette fonction ?

3 Solutions

Solution de l'exercice 1 -

```
def facto(n):
    f = 1
    for k in range(1, n+1):
        f = f * k
    return f
```

Il y a n multiplications.

Solution de l'exercice 2 -

```
def binom_f(n, p):
    return facto(n)//facto(p)//(facto(n-p)
```

Il y a $n+p+n-p=2n$ multiplications et deux divisions entières, c'est un $\mathcal{O}(n)$.

Solution de l'exercice 3 - $\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$

```
def binom_p(n, p):
    b = 1
    for k in range(p):
        b = b * (n-k) // (k+1)
    return b
```

Solution de l'exercice 4 - Les factorielles créent des nombres qui peuvent être très grands, la division peut alors être impossible.

Solution de l'exercice 5 -

```
def liste_binom1(n):
    coefs = [0]*(n+1)
    for p in range(n+1):
        coefs[p] = binom_f(n, p)
    return coefs
```

On fait $n+1$ fois $2n+2$ opérations, soit $2(n+1)^2 = \mathcal{O}(n^2)$.

Solution de l'exercice 6 -

```
def liste_binom2(n):
    fact = [1]*(n+1)
    for p in range(1, n+1):
        fact[p] = fact[p-1]*p
    coefs = [0]*(n+1)
    for p in range(n+1):
        coefs[p] = fact[n] // fact[p] // fact[n-p]
    return coefs
```

Solution de l'exercice 7 -

```
def liste_binom3(n):
    coefs = [1]*(n+1)
    for p in range(n):
        coefs[p+1] = coefs[p] * (n-p) // (p+1)
    return coefs
```

La fonction effectue n multiplications et n additions.

Solution de l'exercice 8 -

```
def pascal(n):
    C = [[]]*(n+1)
    for q in range(n+1):
        C[q] = [1]*(q+1)
        for p in range(1, q):
            C[q][p] = C[q-1][p-1] + C[q-1][p]
    return C
```

La boucle d'indice p effectue $q-1$ additions (pour $q \geqslant 2$) : il y a $\frac{n(n-1)}{2}$ additions.

Solution de l'exercice 9 - `pascal(n)[n]`

Solution de l'exercice 10 - L'idée est de remplacer le terme d'indice p par la somme des termes d'indices p et $p-1$ en décroissant de $q-1$ jusqu'a 1.

```
def liste_binom4(n):
    coefs = [1]*(n+1)
    for q in range(2, n+1):
        for p in range(q-1, 0, -1):
            coefs[p] = coefs[p] +coefs[p-1]
    return coefs
```

Solution de l'exercice 11 -

```
def bernstein(n, f, x):
    C = liste_binom3(n)
    y = 0
    for x in range(n+1):
        y = y + C[k]*f(k/n)*x**k*(1-x)**(n-k)
    return y
```

Solution de l'exercice 12 - La somme des coefficients est 2^{2p}.

Solution de l'exercice 13 - On peut lisser en i et en $n-i-1$ avec $2i+1$ points pour $0 \leqslant i < p$: $widehatx_i = \frac{1}{2^{2i}} \sum_{j=0}^{2i} \binom{2i}{j} x_j$ et $widehatx_{n-1-i} = \frac{1}{2^{2i}} \sum_{j=n-1-2i}^{n-1} \binom{2i}{n-1-j} x_j$ et

Solution de l'exercice 14 -

```
def lissage(x, p):
    n = len(x)
    liss = [0]*n
    coefs = liste_binom3(n)
    for i in range(n):
        if i >= p and i < n - p:
            for j in range(i-p, i+p+1):
                liss[i] = liss[i] + x[j]*coefs[j-i+p]
        else:
            liss[i] = x[i]
    return liss
```

Solution de l'exercice 15 -
On compte le nombre d'additions et de multiplications : on en fait $(n-2p)2(2p+1)$

DS II

VAGUES ET STRUCTURES

Ce sujet est une adaptation du sujet des Mines 2020

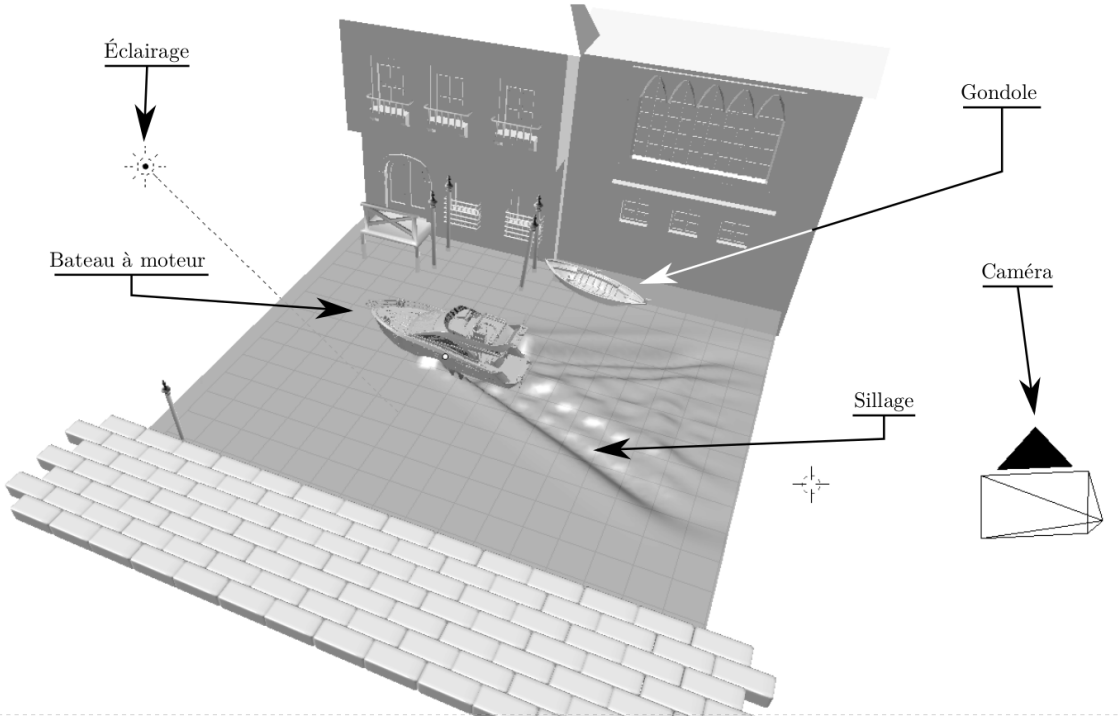


Figure II.1 – Présentation de la scène étudiée

Préambule

Dans une production cinématographique en images de synthèse, les images sont crées une à une pour donner l'illusion du mouvement (sur le principe du dessin animé). Pour satisfaire les spectateurs, il est efficace de réaliser des images conformes aux équations de la physique.
Le sujet aborde la réalisation d'une scène montrant un bateau à moteur traversant un canal, créant un sillage à la surface de l'eau, et faisant osciller une gondole amarrée à proximité.
Il n'est pas nécessaire d'avoir réussi à écrire le code d'une fonction pour pouvoir s'en servir dans une autre question.

Partout où cela est nécessaire, les variables sont considérées être exprimées dans les unités SI.
Le repère $(O, \vec{e}_x, \vec{e}_y, \vec{e}_z)$ servant de référentiel est fixe par rapport au décor.

Modèle de facettes

La scène contient plusieurs objets géométriques tri-dimensionnels (3D). Chaque objet géométrique est représenté de manière numérique par un maillage.

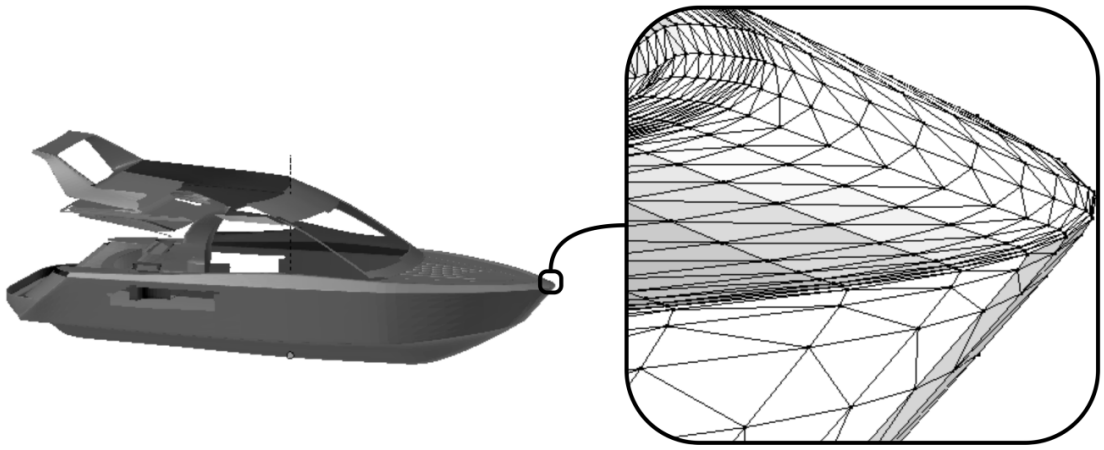


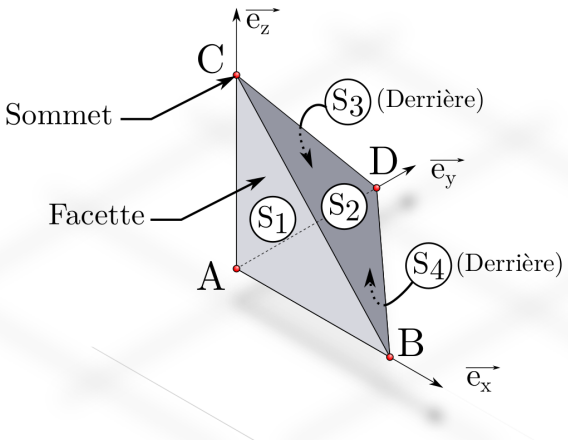
Figure II.2 – Un exemple de maillage : la coque d'un bateau

On définit les termes suivants :

Maillage : ensemble des facettes qui constituent la géométrie d'un objet. Un maillage sera représenté par une liste de facettes.

Facette : polygone élémentaire qui constitue une partie de la surface d'un objet. Ici, toutes les facettes seront des triangles. Une facette sera représentée par une liste ordonnée de 3 sommets.

Sommet : point délimitant une facette. Il peut être commun à une ou plusieurs facettes. Tout point sera représenté par son vecteur position de coordonnées (x, y, z).



La figure ci-contre représente un exemple de maillage simple (tétraèdre),
composé de 4 facettes (notées S_1 à S_4)
et de 4 sommets (notés A à D)
avec $AB = AD = AC = 1$.

Numpy

On utilisera les vecteurs (`array`) du module `numpy`.

```
import numpy as np
```

Par exemple, un vecteur dont les coordonnées sont $x = 2$, $y = 3$ et $z = 1,5$ sera defini par :

```
V = np.array([2.0, 3.0, 1.5])
```

On rappelle qu'il est affiché par `array([2.0, 3.0, 1.5])`.

- Un sommet point est représenté par un vecteur `numpy`, A est assimilé à $\overrightarrow{OA}$.
- Si `a` et `b` représentent les points A et B, le vecteur $\overrightarrow{AB}$ est déterminé par `b - a`.
- On peut additionner les vecteurs `numpy` de même taille.
- On peut multiplier un vecteur par un scalaire.
- Le produit scalaire $\vec{u}.\vec{v}$ de deux vecteurs représentés par les vecteurs `numpy` `u` et `v` est obtenu par `np.dot(u, v)`.
- Le produit vectoriel $\vec{u} \wedge \vec{v}$ est calculé par `np.cross(u, v)`.

Représentation Python

- Une facette triangulaire est représentée par la liste de ses 3 sommets.
- Un maillage est représenté par la liste de ses facettes.

Par exemple, le tétraèdre ci-dessus sera défini par la variable `maillage_tetra`

```
A = np.array([0.0, 0.0, 0.0])
B = np.array([1.0, 0.0, 0.0])
C = np.array([0.0, 0.0, 1.0])
D = np.array([0.0, 1.0, 0.0])

maillage-tetra = [[A, B, C], [A, D, C], [A, B, D], [B, D, C]]
```

1 Objets de la scène

1.1 Travail sur les facettes

Exercice 1
À partir de la variable `maillage_tetra`*, écrire une expression Python permettant de récupérer la coordonnée y du premier sommet de la première facette.*

Exercice 2
À quel élément, sur la figure de la page 2, correspond `maillage_tetra[1]` *?*

On considère la fonction suivante, prenant un vecteur comme paramètre

```
def mystere1 (V):
    x, y, z = V
    return (x**2 + y**2 + z**2)**0.5
```

Exercice 3
Que fait la fonction `mystere1` *?*
Proposer un nom plus adapté pour la fonction ; on pourra utiliser la fonction avec ce nouveau nom.

Exercice 4
Écrire une fonction `barycentre(F)`*, prenant comme argument une facette* `F` *(une liste de 3 sommets) et renvoyant le barycentre de la facette, c'est-à-dire le point dont les coordonnées sont les moyennes des coordonnées des sommets.*

On admet que l'aire d'un triangle (ABC) est calculée par $\mathcal{A}(ABC) = \left\|\overrightarrow{AB} \wedge \overrightarrow{AC}\right\|$.

Exercice 5
Écrire une fonction `aire(F)`*, prenant comme argument une facette* `F` *et renvoyant son aire.*

Le **vecteur unitaire normal** d'une facette `F = [A, B, C]` d'aire non nulle est calculé par

$$\vec{n} = \frac{\overrightarrow{AB} \wedge \overrightarrow{AC}}{\left\|\overrightarrow{AB} \wedge \overrightarrow{AC}\right\|}$$

Exercice 6
Écrire une fonction `normal(F)`*, prenant comme argument une facette* `F` *et renvoyant le vecteur unitaire normal.*

1.2 Liste des sommets

Compte tenu de la représentation limitée des nombres réels en machine, deux sommets S_1 et S_2 supposés être au même endroit peuvent avoir des coordonnées légèrement différentes.

Exercice 7
Proposer une fonction `sont_proches(S1, S2, epsilon)`*, prenant comme arguments deux sommets* `S1` *et* `S2` *et un flottant strictement positif* `epsilon`*, et qui renvoie* `True` *si* S_1 *et* S_2 *sont proches (i.e. si leur distance au sens de la norme Euclidienne est inférieure à* ε*) et* `False` *sinon.*

Soient les fonctions suivantes :

```
def mystere2(S1, L):
    for S2 in L :
        if sont_proches (S1, S2, 1e-7):
            return True
    return False
```

```
def mystere3(maillage):
    res = []
    for facette in maillage:
        for sommet in facette:
             if not mystere2(sommet, res):
                 res.append(sommet)
    return res
```

Exercice 8
Sous quelle condition la fonction `mystere2` *renvoie-t-elle* `True` *?*

Exercice 9
Donner (sans justification) ce que renvoie `mystere3(maillage_tetra)`*, où* `maillage_tetra` *est la variable définie précédemment.*

Exercice 10
Pour une liste `L` *de longueur* n*, discuter la complexité de la fonction* `mystere2`*. En déduire la complexité de* `mystere3`*, pour un maillage contenant* m *facettes triangulaires. On distinguera le meilleur et le pire des cas.*

2 Mouvement de flottaison

La gondole amarrée sur le bord du canal perçoit les vagues générées par le bateau à moteur et effectue un mouvement pseudo-oscillant conséquence de sa flottaison. On étudie ici le mouvement de translation verticale de la gondole (on ne prendra en compte ni le tangage ni le roulis). On cherche à modéliser ce mouvement en calculant à un instant donné la force exercée par le fluide sur la gondole.

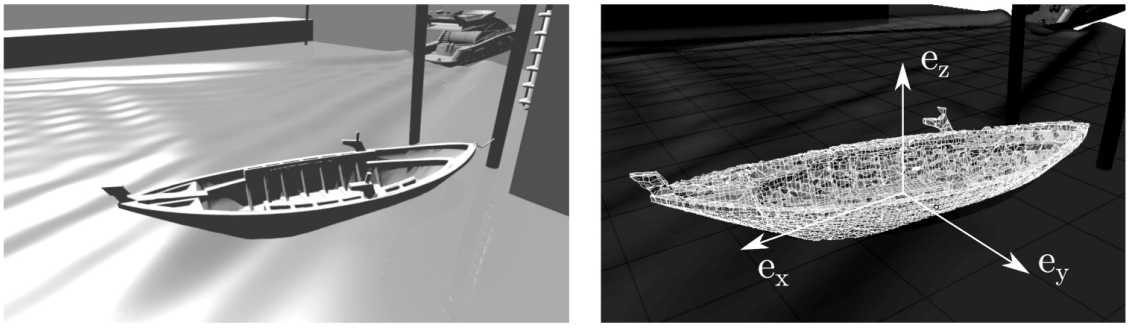


Figure II.3 – Maillage de la gondole

Seul le mouvement de translation verticale (selon la direction verticale $\vec{e}_z$) est étudié.
On s'intéresse ici au maillage qui constitue la coque extérieure de la gondole. Certaines facettes sont émergées (i.e. leur barycentre est en dehors de l'eau), d'autres sont immergées (i.e leur barycentre est sous l'eau).
À chaque pas de temps, on suppose connue la fonction `hauteur(x,y)` qui, à tout point (x, y) du plan d'eau, associe la hauteur des vagues par rapport au repère de la scène. Le maillage composant la coque de la gondole est défini dans une liste de facettes nommée `maillageG`.

Exercice 11
Proposer une fonction `lister_FI(M)` *prenant comme argument un maillage* `M` *et renvoyant la liste des facettes immergées (i.e dont le centre de gravité est sous la surface définie par hauteur). On pourra utiliser les fonctions de la première partie.*

Pour calculer la poussée d'Archimède s'exerçant sur la coque du bateau, on doit calculer la résultante des forces dues à la pression appliquées par l'eau sur chacune des facettes immergées.
On modélise la force appliquée par l'eau sur une facette i par :

$$\overrightarrow{F_i} = -S_i \times p(G_i)\vec{n}_i$$

- S_i est l'aire de la facette,
- $\vec{n}_i$ est vecteur normal sortant de la coque,
- $p(G_i)$ est la pression hydrostatique de l'eau sur la facette en son barycentre G_i.

Le théorème de Pascal détermine la pression de l'eau d'un point G en fonction de sa profondeur par rapport à la surface :

$$p(x_G, y_G, z_G) = \rho \cdot g \cdot \big(\texttt{hauteur}(x_G, y_G) - z_g\big)$$

- ρ est masse volumique de l'eau (~ 1000),
- g est l'accélération de la pesanteur (ici : $g \sim 9,81$).

Exercice 12
Proposer une fonction `force_facette(F)` *prenant en argument une facette et renvoyant le vecteur force appliqué par l'eau sur cette facette.*

On pourra utiliser les fonctions définies précédemment et on supposera que les sommets sont donnés dans les facettes dans un ordre tel que le vecteur normal calculé est sortant.

La force résultante sur toute la coque s'exprime par la somme de toutes les forces appliquées sur chaque facette immergée.

Exercice 13
Définir la fonction `resultante(L)` *prenant comme argument une liste* `L` *de facettes (supposées immergées), renvoyant la somme des forces, sur l'axe* $\vec{e}_z$ *de l'eau, appliquée sur l'ensemble des surfaces.*

On cherche à optimiser l'efficacité de la fonction `resultante` qui devra être utilisée intensément pour réaliser un grand nombre d'images. On remarque que la taille des facettes n'est pas homogène : la coque est composée de grandes facettes et de petites facettes. Les petites facettes représentent souvent des détails d'intérêt graphique n'apportant qu'une très faible contribution à la résultante des forces hydrostatiques.
Ainsi, une étude montre que la moitié des facettes représente à elle seule 99% de la surface totale de la coque. Pour alléger le processus, on trier les facettes par aire décroissante, afin de n'appliquer les calculs de la poussée d'Archimède qu'à la moitié d'entre elles (les plus grandes). La fonction de tri est supposée donnée sous le nom `trier_facettes`.

Exercice 14
Affecter à une nouvelle variable `grandesFacettes` *la liste des facettes de privée de la moitié des facettes les plus petites (en cas de nombre impair d'éléments, on inclura la facette médiane).*

La gondole est attachée à un repère local dont l'origine est son centre de gravité (de coordonnées (x_G, y_G, z_G) et de vitesse verticale notée v) par rapport au décor. Le principe fondamental de la dynamique en projection sur l'axe vertical $\vec{e}_z$ appliqué à la gondole énonce que

$$\begin{cases} \dfrac{\mathrm{d}v}{\mathrm{d}t} &= \frac{1}{m} \times F_g - g \\ \dfrac{\mathrm{d}z_G}{\mathrm{d}t} &= v \end{cases}$$

- m est la masse de la gondole,
- F_g est la résultante des forces appliquées par l'eau sur la gondole (renvoyée par la fonction `resultante`, vue précédemment).

La position initiale de la gondole est $z_{G,0} = 0$. Sa vitesse verticale initiale est $v_0 = 0$.
On souhaite estimer le mouvement par la méthode d'Euler. Pour ce faire, on utilise la fonction `nouvelle_hauteur` avant d'afficher chaque nouvelle image. Cette fonction a pour but de recalculer la hauteur (et la vitesse) de la gondole pour un nouveau pas de temps. Elle prend trois arguments :

- `posG` contiendra le vecteur position actuel du centre de gravité de la gondole au moment de l'appel,
- `vitG` contiendra le vecteur vitesse actuel de ce même point au moment de l'appel,
- `mailG` ontiendra la liste des grandes facettes de la gondole (privée des petites, au sens de la question précédente), au moment de l'appel.

```
def nouvelle_hauteur(posG, vitG, mailG):
    dt = 1/25 # Pas de temps correspondant à une image du film
    facettes_immergees = lister_FI(mailG)
    posG = posG + .......... # à compléter
    vitG = vitG + .......... # à compléter
    return posG, vitG
```

Exercice 15
Compléter les lignes 4 et 5 du code précédent conformément à la méthode d'Euler.

3 Solutions

Solution de l'exercice 1 -

```
maillage_tetra[0][0][1]
```

Solution de l'exercice 2 - La facette (A, D, C) correspond à la facette S_3.

Solution de l'exercice 3 -
La fonction calcule la norme du vecteur V. On pouvait l'appeler `norme`.

Solution de l'exercice 4 -

```
def barycentre(F):
    A, B, C = F
    return (A + B + C)/3
```

Solution de l'exercice 5 -

```
def aire(F):
    A, B, C = F
    N = np.cross(B-A, C-A)
    return norme(N)
```

Solution de l'exercice 6 -

```
def normal(F):
    A, B, C = F
    N = np.cross(B-A, C-A)
    return (1/norme(N))*N
```

Solution de l'exercice 7 -

```
def sont_proches(S1, S2, epsilon):
    distance = norme(S2 - S1)
    return distance < epsilon
```

Solution de l'exercice 8 - La fonction `mystere2` renvoie `True` si le sommet `S1` est proche de l'un des sommets de la liste `L`, avec une précision de 10^{-7}.

Solution de l'exercice 9 - `mystere3(maillage_tetra)` renvoie les sommets distincts à 10^{-7} près du maillage. Ici la fonction renvoie

```
[array([0., 0., 0.]), array([0., 0., 1.]),
 array([0., 1., 0.]), array([1., 0., 0.])]
```

Solution de l'exercice 10 - La fonction `mystere2(S1, L)` fait un nombre constant d'instructions par passage dans la boucle.
Le meilleur cas est celui où `S1` est proche du premier élément de `L`, ce qui n'occasionne qu'un passage dans la boucle, d'où une complexité en $\mathcal{O}(1)$.
Au pire `S1` n'est proche d'aucun élément de la liste est la boucle est parcourue n fois d'où une complexité en $\mathcal{O}(n)$.

Pour la fonction `mystere3(maillage)`, on applique la fonction `mystere2(S, res)` pour tous les sommets du maillage, il y en a $3m$.

Dans le meilleur des cas, tous les sommets sont proches du premier et la complexité est alors proportionnelle à m : $\mathcal{O}(m)$.
Dans le pire cas, aucun sommet n'est proche d'un autre sommet. La liste `res` est de longueur $k-1$ lors du traitement du k-ième sommet d'où une complexité majorée par
$\sum_{k=1}^{3m} A.(k-1) = A\frac{3m(3m-1)}{2} = \mathcal{O}(m^2)$.

Solution de l'exercice 11 -

```
def lister_FI(M):
    L = []
    for facette in M:
        x, y, z = barycentre(facette)
        if z < hauteur(x, y):
            L.append(facette)
    return L
```

Solution de l'exercice 12 -

```
def force_facette(F):
    S = aire(F)
    n = normale(F)
    x, y, z = barycentre(F)
    p = rho*g*(hauteur(x,y) - z)
    return -S*p*n
```

Solution de l'exercice 13 -

```
def resultante(L):
    res = 0.
    for F in L:
        res = res + force_facette(F)[2]
    return res
```

Solution de l'exercice 14 -

```
n = len((maillageG)
p = (n + 1)//2
facettes_triees = trier_facettes(maillageG)
grandesFacettes = facettes_triees[ : p]
```

Solution de l'exercice 15 -

```
def nouvelle_hauteur(posG, vitG, mailG):
    dt = 1/25
    facettes_immergees = lister_FI(mailG)
    posG = posG + dt*vitG
    vitG = vitG + dt*(resultante(facettes_immergees)/m - g)
    return posG, vitG
```

DS III

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 considérerons 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, deux ou trois dimensions ; le récipient contenant le gaz sera, suivant le cas, un segment de longueur L, un carré de côté L ou un cube d'arête L.

Conventions

- Pour répondre à une question il est possible de faire appel aux fonctions définies dans les questions précédentes.
- Dans tout le sujet on suppose que les bibliothèques `math`, `numpy` et `random` ont été importées grâce aux instructions

```
import math
import random
```

- Ce sujet utilise une syntaxe raccourcie pour les instructions `if` suivie d'une seule instruction courte. L'écriture

```
    if a < 1:
        return True
```

pourra être écrite sous la forme

```
    if a < 1: return True
```

- Ce sujet utilise la syntaxe des annotations pour préciser le types des arguments et du résultat des fonctions à écrire. Ainsi

```
def maFonction(n:int, x:float, l:[str]) -> (int, [int]):
```

signifie que la fonction `maFonction` prend trois arguments, le premier est un entier, le deuxième un nombre à virgule flottante et le troisième une liste de chaînes de caractères et qu'elle renvoie un couple dont le premier élément est un entier et le deuxième une liste d'entier. **Il n'est pas demandé aux candidats de recopier les entêtes avec annotations telles qu'elles sont fournies dans ce sujet**, ils peuvent utiliser des entêtes classiques.

```
def maFonction(n, x, l):
```

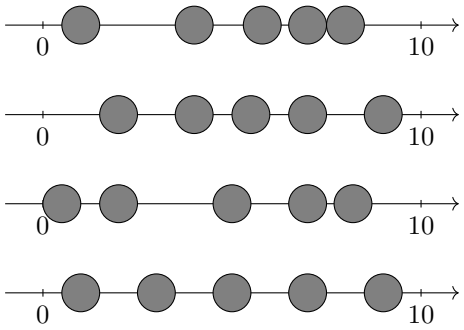
Les candidats veilleront cependant à décrire précisément le rôle des fonctions qu'ils définissent.
- `random.random()` renvoie un nombre flottant tiré aléatoirement dans $[0, 1[$ suivant une distribution uniforme

1 Initialisation

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 et dotées d'une vitesse initiale connue. Cette partie s'intéresse au positionnement aléatoire d'un ensemble de particules.

1.1 Placement en dimension 1

Nous cherchons d'abord comment 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 $L = 10$.



La fonction `placement1D` construit aléatoirement, à partir des paramètres géométriques du problème (nombre et rayon des particules, taille du récipient), une liste de coordonnées correspondant à la position initiale du centre de chaque particule.

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

Exercice 1

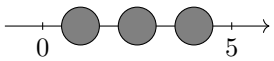
1. *Détailler l'action de la ligne 9.*
2. *Quelle est la signification du paramètre* `c` *de la fonction* `possible` *(ligne 2) ?*
3. *Expliquer le rôle de la ligne 3.*
4. *Expliquer le rôle des lignes 4 et 5.*
5. *Donner en une phrase le rôle de la fonction* `possible`*.*

Exercice 2

Proposer une nouvelle version de la ligne 9 permettant d'éviter certains rejets de la part de la fonction `possible`*.*

Exercice 3

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, 1,5 et 4 (voir ci-contre).*
Quelle sera la suite du déroulement de la fonction `placement1D` *?*



Exercice 4
Quelle est la complexité temporelle[1] *de la fonction* `placement1D` *dans le cas*[2] *où* $N \ll N_{\max}$, *nombre maximal de particules de rayon* R *pouvant être placées sur un segment de longueur* L *?*

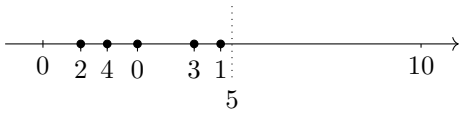
Exercice 5
Pour remédier de manière simple à la situation de la question 3, on décide de recommencer à zéro le placement des particules dès qu'une particule est rejetée par la fonction `possible`.
Réécrire les lignes 7 à 11 de la fonction `placement1D` *pour mettre en œuvre cette décision.*

1.2 Optimisation du placement en dimension 1

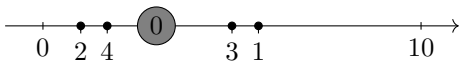
Pour placer aléatoirement N particules le long d'un segment, nous envisageons une approche plus efficace que celle étudiée dans la partie précédente.
L'idée est de calculer l'espace laissé libre sur le segment cible par N particules[3] puis de répartir aléatoirement cet espace libre entre les particules. Afin de conserver une répartition uniforme des particules dans tout le segment, nous utilisons l'algorithme suivant :

1. déterminer ℓ, espace laissé libre par les N particules dans le segment $[0, L[$;
2. placer aléatoirement dans le segment $[0, \ell[$, N particules virtuelles ponctuelles ($R = 0$) ;
   à cette étape, deux particules peuvent occuper la même abscisse : il n'y a pas de conflit ;
3. 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.

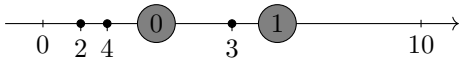
Exemple : $L = 10$, $R = 0.5$, $N = 5$.
On choisit les positions :



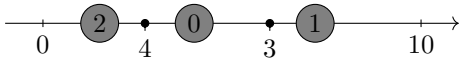
On crée la particule 0 et on déplace les suivantes :



On crée la particule 1 et on déplace les suivantes :

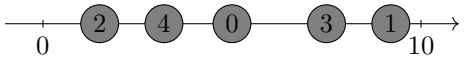


On crée la particule 2 et on déplace les suivantes :



On crée la particule 3 et on déplace les suivantes :

On crée la particule 4 et on déplace les suivantes :



1. On comptera le nombre de comparaisons.
2. On admettra que, dans ce cas, la fonction `possible(p)` renvoie `True` à chaque appel.
3. C'est-à-dire La longueur totale diminuée du diamètre de toutes les particules.

Exercice 6
Écrire la fonction `placement1Drapide(N, R, L)` *qui implante cet algorithme et renvoie la liste des coordonnées des centres de* `N` *particules de rayon* `R` *réparties aléatoirement le long d'un segment situé entre les abscisses* 0 *et* `L`.
On précise que l'ordre de la liste résultat n'est pas important.

Exercice 7
Quelle est la complexité de la fonction `placement1Drapide` *? Commenter.*

1.3 Utilisation d'un tri

On suppose maintenant qu'entre l'étape 2 et l'étape 3 on effectue le tri des position par ordre croissant par le tri-fusion.
Dans l'exemple illustré ci-dessus, la liste des positions initiales était `[2.5, 4.7, 1.0, 4.0, 1.7]` et le tri fournit la liste `[1.0, 1.7, 2.5, 4.0, 4.7]`.
On supposera écrite une fonction `tri_fusion(liste)` qui renvoie une liste qui comporte les mêmes éléments que la liste passée en paramètre mais placé en ordre croissant.
On ne demande pas d'écrire cette fonction

Exercice 8
Donner rapidement le principe du tri-fusion. On n'expliquera pas la fusion.
Quelle est sa complexité ?

Exercice 9
Écrire la fonction `placement1DplusRapide(N, R, L)` *qui implante l'algorithme ci-dessus en insérant le tri et en exploitant le caractère trié de la liste pour effectuer l'étape avec une complexité en* $\mathcal{O}(N)$.
Quelle est maintenant la complexité de la fonction ?

1.4 Dimension quelconque

L'algorithme optimisé pour un segment, n'est pas utilisable pour des espaces de dimensions supérieures. Nous allons donc généraliser la fonction `placement1D` pour la transformer en une fonction utilisable dans un espace de dimension 1, 2 ou 3.

Exercice 10
En s'inspirant de la fonction `placement1D`*, écrire la fonction d'entête*

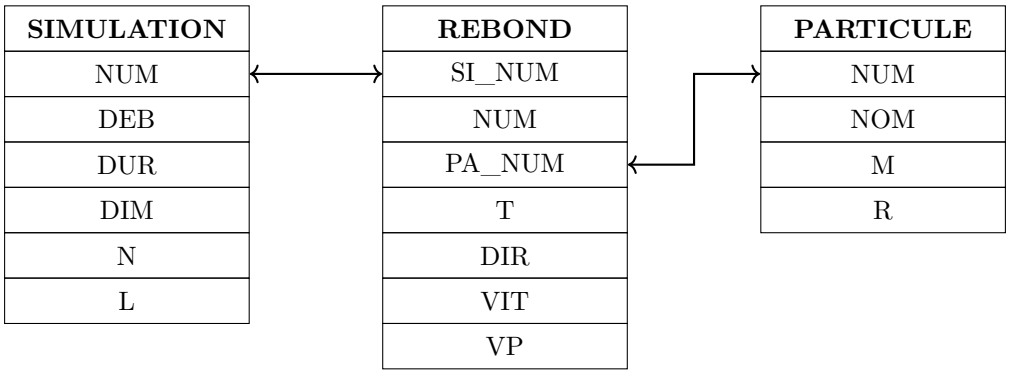
```
def placement(D:int, N:int, R:float, L:float) -> [[float]]:
```

qui renvoie la liste des coordonnées des centres[4] *de* `N` *particules sphériques de rayon* `R` *placées aléatoirement dans un récipient de côté* `L` *dans un espace à* `D` *dimensions.*
Les modifications prévues aux questions 2 et 5 seront prises en compte dans cette fonction.

4. Sous forme de listes de D réels.

2 Exploitation des résultats

On simule le mouvement de particules en comptant le nombre de chocs sur les parois du contenant. On généralise les calculs en ne supposant plus que les particules sont identiques. Les résultats sont enregistrés dans base de données à trois tables :



SIMULATION

C'est la table qui donne les caractéristiques de chaque simulation effectuée :
- `NUM` numéro d'ordre de la simulation (clef primaire),
- `DEB` date et heure du lancement du programme de simulation,
- `DUR` durée (en secondes) de la simulation,
- `DIM` nombre de dimensions de l'espace de simulation,
- `N` nombre de particules pour cette simulation,
- `L` (en mètres) taille du récipient utilisé pour la simulation.

PARTICULE

C'est la table qui donne les types de particules considérées :
- `NUM` numéro (entier) identifiant le type de particule (clef primaire).
- `NOM` nom de ce type de particule.
- `M` masse de la particule (en grammes).
- `R` rayon (en mètres) de la particule.

REBOND

Cette table liste les chocs des particules avec les parois du récipient.
- `SI_NUM` numéro d'ordre de la simulation ayant généré ce rebond.
- `NUM` numéro d'ordre du rebond au sein de cette simulation.
- `PA_NUM` numéro du type de particule concernée par ce rebond.
- `T` temps de simulation (en secondes) auquel ce rebond est arrivé.
- `DIR` paroi concernée : entier non nul de l'intervalle `[-DIM, DIM]` donnant la direction de la normale à la paroi. Ainsi -1 désigne la paroi située en $x = 0$, 1 désigne la paroi située en $x = L$, -2 désigne la paroi située en $y = 0$. ...
- `VIT` norme de la vitesse de la particule qui rebondit.
- `VP` valeur absolue de la composante de la vitesse normale à la paroi.

Exercice 11
Proposer une clef primaire pour la table `REBOND`*, elle comporte plus d'une colonne.*

Exercice 12
Écrire une requête SQL qui donne le nom des particules qui ont une masse supérieure à 5.10^{-23} *grammes (*`5e-23`*).*

Exercice 13
Écrire une requête SQL qui donne, pour chaque simulation, le nombre (`count`*) de rebonds enregistrés et la vitesse moyenne (*`avg`*) des particules qui frappent une paroi.*

Exercice 14
Quelles sont les noms des particules qui sont intervenues dans la simulation de numéro 1472 ?

Exercice 15
Quelles sont les durées des simulation qui ont simulé plus de 100 000 rebonds ?

Exercice 16
Écrire une requête SQL qui, pour la simulation de numéro 4782*, calcule, pour chaque paroi, la variation de quantité de mouvement due aux chocs des particules sur cette paroi tout au long de la simulation (*`sum`*).*
On se rappellera que lors du rebond d'une particule sur une paroi la composante de sa vitesse normale à la paroi est inversée, ce qui correspond à une variation de quantité de mouvement de $2m\|v_\perp\|$ *où* m *désigne la masse de la particule et* $v_\perp$ *la composante de sa vitesse normale à la paroi.*

3 Solutions

Solution de l'exercice 1 -

1. À la ligne 9, on affecte à `p` un réel aléatoire entre 0 et L.
2. Le paramètre `c` est la position possible d'une particule.
3. À la ligne 3, on vérifie que la particule est tout entière entre les parois : son centre doit avoir une position entre R et $L - R$.
4. Les lignes 4 et 5 vérifient que la nouvelle particule n'essaie pas d'occuper une position déjà prise.
5. `possible(c)` teste donc si `c` est une position possible pour une nouvelle particule, compte-tenu de celles déjà présentes.

Solution de l'exercice 2 - On peut éviter de tester l'inclusion entre les parois en choisissant le centre entre R et $L - R$:

```
def placement1D(N, R, L):

    def possible(c:float) -> bool:
        for p in res:
            if abs(c - p) < 2*R: return False
        return True

    res = []
    while len(res) < N:
        p = R + (L - 2*R)*random.random()
        if possible(p): res.append(p)
    return res
```

Solution de l'exercice 3 -
Avec les trois premières particules placées, il ne reste plus d'espace de largeur 1 disponible.
Quel que soit la valeur de p, `possible(p)` va renvoyer `False` et la boucle `while` tournera sans fin.

Solution de l'exercice 4 - On suppose donc que la boucle `while` est effectuée N fois.
À chaque passage on compare la position avec celles des particules déjà placées on fait donc $0 + 1 + 2 + \cdots + (N-1) = \frac{N(N-1)}{2}$ comparaisons la complexité est un $\mathcal{O}(N^2)$.

Solution de l'exercice 5 -

```
def placement1D(N, R, L):

    def possible(c:float) -> bool:
        for p in res:
            if abs(c - p) < 2*R: return False
        return True

    res = []
    while len(res) < N:
        p = R + (L - 2*R)*random.random()
        if possible(p):
            res.append(p)
        else:
            res = []
    return res
```

Solution de l'exercice 6 - On doit écarter toutes les boules qui suivent de la valeur d'un diamètre mais aussi écarter d'un rayon la boule actuelle.

```
def placement1Drapide(N, R, L):
    # Etape 1
    l = L - 2*N*R
    # Etape 2
    res=[]
    for i in range(N):
        res.append(l*random.random())
    #Etape 3
    for i in range(N):
        pos=res[i]
        for j in range(N):
            if res[j]>= pos and j !=i :
                res[j] = res[j] + 2*R
        res[i] = res[i] + R
    return res
```

Solution de l'exercice 7 - Les deux premières étapes se font en temps linéaire en fonction de N. Pour la troisième étape, les deux boucles imbriquées induisent un nombre d'opérations de l'ordre de N^2.
On a encore une complexité en $\mathcal{O}(N^2)$ mais il n'y a plus le risque d'une complexité plus importante en cas densité de particules trop grande.

Solution de l'exercice 8 - Le tri fusion sépare la liste en deux partie de tailles quasi-égales, trie récursivement les deux partie puis fusionne les deux listes triées. Ce travail n'est effectué que pour les listes de taille 2 au moins.
Sa complexité est un $\mathcal{O}\big(\log(N)\big)$.

Solution de l'exercice 9 -

```
def placement1Drapide(N, R, L):
    # Etape 1
    l = L - 2*N*R
    # Etape 2
    res=[]
    for i in range(N):
        res.append(l*random.random())
    # Tri
    res = tri_fusion(res)
    # Etape 3
    for i in range(N):
        res[i] = res[i] + R + 2*i*R
    return res
```

La complexité est celle de la portion la plus lente, c'est un $\mathcal{O}\big(\log(N)\big)$.

Solution de l'exercice 10 - On écrit une fonction `distance` qui calcule la distance entre deux particules.

```
def distance(p1, p2):
    n = len(p1) # aussi len(p2)
    d = 0
    for i in range(n):
        d = d + (p1[i] - p2[i])**2
    return d**(1/2)
```

Il reste à adapter la fonction `placement` en créant des particules de bonnes dimensions.

```
def placement(D, N, R, L):
    def possible(c):
        for p in res:
            if distance(c, p) < 2*R:
                return False
        return True

    res=[]
    while len(res) < N:
        p = [0]*D
        for i in range(D):
            p[i] = R + random.random()*(L-2*R)
        if possible(p):
            res.append(p)
        else:
            res=[]
    return res
```

Solution de l'exercice 11 - On doit différentier les rebonds de chaque simulation : `(SI_NUM, NUM)` est une clef primaire.

Solution de l'exercice 12 -

```
select NOM
from PARTICULE
where M > 5e-23
```

Solution de l'exercice 13 -

```
select SI_NUM, count(), avg(VIT)
from REBOND
group by SI_NUM
```

Solution de l'exercice 14 -

```
select distinct p.NOM
from REBOND as r join PARTICULE as p on r.PA_NUM = p.NUM
where r.SI_NUM = 1472
```

Solution de l'exercice 15 -

```
select r.SI_NUM, count() as nb, s.DUR
from REBOND as r join SIMULATION as s
group by r.SI_NUM
having nb > 1e5
```

Solution de l'exercice 16 -

```
select r.DIR, sum(2*p.M*r.VP)
from REBOND as r join PARTICULE as p on  r.PA_NUM = p.NUM
WHERE SI_NUM = 4782
group by DIR
```

DS IV

Centrale 2017

1 Création d'une exploration et gestion des points d'intérêt

1.1 Génération d'une exploration d'essai

Choix de points au hasard

Exercice 1 — I.A.1) a)
On choisit des points au hasard mais ils ne doivent pas avoir déjà été choisis. Le formulaire permet d'utiliser `c in liste` pour tester l'appartenance d'un élément à une liste ; si les points ne sont pas dans la liste des points déjà choisis, on les ajoute et on incrémente le compteur de points. Il ne faut pas oublier de convertir en tableau `numpy` à la fin.
Un tableau `numpy` ne peut pas être défini à l'aide d'ajouts (`append`) or on ne sait pas combien il y aura d'éléments. On définit donc une liste python que l'on convertit lors du retour de la fonction.

```
def generer_PI(n, cmax):
    points = []
    nb_points = 0
    while nb_points < n:
        x = random.randrange(0, cmax)
        y = random.randrange(0, cmax)
        if [x, y] not in points:
            points.append([x, y])
            nb_points += 1
    return np.array(points, dtype = int)
```

Exercice 2 — I.A.1) b)
On choisit n points parmi `cmax`^{2} points possibles donc on doit avoir $n \leqslant cmax^2$.

Calcul des distances

Exercice 3 — I.A.2)
Il est utile de définir la distance entre deux points.

```
def distance(P1, P2):
    x1, y1 = P1
    x2, y2 = P2
    return ((x1-x2)**2 + (y1-y2)**2)**0.5
```

On ne doit pas oublier d'ajouter la distance à l'origine du robot.

```
def calculer_distances(PI):
    n = len(PI)
    points = np.zeros((n+1, 2), dtype = int)
    for i in range(n):
        points[i] = PI[i]
    x, y = position_robot()
    points[n, 0] = x
    points[n, 1] = y
    d = np.zeros((n+1, n+1))
    for i in range(n+1):
        for j in range(n+1):
            d[i, j] = distance(points[i], points[j])
    return d
```

1.2 Traitement d'image

Analyse d'une image

Exercice 4 — I.B.1)
Cette fonction parcourt tous les pixels de l'image et incrémente un tableau de compteurs à chaque intensité rencontrée.
Elle renvoie donc un tableau contenant le nombre de pixel de chaque intensité rencontrée :
`h[k]` contient le nombre de pixels d'intensité $k + n$ où n est l'intensité minimale.
Je ne comprends pas comment on peut utiliser ce tableau : on ne sait pas à quelles intensités correspondent les valeurs.

Sélection de points d'intérêts

Exercice 5 — I.B.2)

```
def selectionner_PI(photo, imin, imax):
    longueur, hauteur = photo.shape
    L = []
    for x in range(longueur):
        for y in range(hauteur):
            if imin <= photo[x, y] <= imax:
                L.append([x, y])
    return np.array(L)
```

1.3 Base de données

Exercice 6 — I.C.1)

```
select EX_NUM
from EXPLO
where EX_DEB is not null and EX_FIN is null
```

Exercice 7 — I.C.2)
Par exemple pour le numéro 42

```
select PI_NUM, PI_X, PI_Y
from PI
where EX_NUM = 42;
```

Exercice 8 — I.C.3)

```
select p.EX_NUM, (max(p.PI_X) - min(p.PI_X)) * (max(p.PI_Y) -
   min(p.PI_Y))/1000000.0
from PI as p join EXPLO as e on p.EX_NUM = e.EX_NUM
where e.EX_DEB is not null and e.EX_FIN is not null
group by p.EX_NUM
```

Exercice 9 — I.C.4)
On ne peut pas répondre à cette question car on ne sait pas comment sont codés les entiers dans le SGBD ... On suppose que les entiers sont codés sur 64 bits, il valent donc au plus $2^{63}-1$ donc la surface maximale sera $\left(2^{63}-1\right)^2 \cdot 10^{-6}$ m^2 soit environ 6.10^{25} km^2, ce n'est pas vraiment une limite.

Exercice 10 — I.C.5)

```
select i.IN_NUM, COUNT(*), SUM(it_dur)
from INTYP as i join ANALY as a on i.TY_NUM = a.TY_NUM
                join EXPLO as e on a.EX_NUM = e.EX_NUM
where e.EX_DEB is not null and e.EX_FIN is null
group by i.IN_NUM
```

2 Planification d'une exploration : première approche

2.1 Quelques fonctions utilitaires

Longueur d'un chemin

Exercice 11 — I.C.5)
Pour calculer la longueur d'un chemin, on commence par la distance entre le point courant et le premier point puis on ajoute les distances entre les points consécutifs.

```
def longueur_chemin(chemin, d):
    n = len(chemin) # d est de taille (n+1)x(n+1)
    longueur = d(n, chemin[0])
    for k in range(n-1):
        longueur += d[chemin[k], chemin[k+1]]
    return longueur
```

Normalisation d'un chemin

Exercice 12 — I.C.5)
On utilise une liste qui repère les sommets déjà vus. On lis les points du chemin et on ajoute ceux qui n'ont pas encore été vus et on les marque comme vus. On ajoute ensuite tous les points non vus.

```
def normaliser_chemin(chemin, n):
    vus = [False]*n
    normal = []
    for i in chemin:
        if 0 < i < n and not vus[i]:
            normal.append(i)
            vus[i] = True
    for j in range(n):
        if not vus[j]:
            normal.append(j)
    return normal
```

2.2 Force brute

Exercice 13 — II.B.1)
Le nombre de chemins est le nombre de permutations des n points d'intérêts, il y en a donc $n!$.

Exercice 14 — II.B.2)
On a $20! \approx 2 \cdot 10^{18}$ si on imagine pouvoir traiter un chemin en 10^{-6} seconde, il faudrait plus de 100 000 ans pour tester tous les chemins. Ce n'est pas praticable.

2.3 Algorithme du plus proche voisin

Exercice 15 — II.C.1)
Comme dans la normalisation, on utilise une liste pour repérer les sommets déjà utilisés.
On rappelle que le point de départ correspond au dernier indice dans la matrice des distances et qu'il n'est pas un point d'intérêt.

```
def plus_proche_voisin(d):
    n = len(d)
    vus = [False]*n
    maxi = d.max() + 1 # Supérieure à toutes les distances
    vus[n-1] = True # Le point de départ est vu
    chemin = [0]*(n-1) # (n-1) n'est pas dans le chemin
    i0 = n-1 # on cherche le plus proche à partir de i0
    for i in range(n-1):
        d_min = maxi # On initialise la distance
        # On cherche  la plus petite distance
        for k = 0 to (n-1): # parmi les points non vus
            if not vus[k] and d[i0, k] < d_min:
                d_min = d[i0, k]
                j0 = k
        # j0 est le bon candidat
        vus[j0] = True
        chemin[i] = j0
        i0 = j0 # il devient le point de départ
    return chemin
```

Exercice 16 — II.C.2)
Le calcul de maxi est linéaire en la taille de la matrice : n^2.
On effectue ensuite une double boucle avec des instructions élémentaires donc la complexité de ces boucles est un $\mathcal{O}(n^2)$.
La complexité totale est aussi un $\mathcal{O}(n^2)$.

Exercice 17 — II.C.3)
Avec les points de coordonnées $A = (0, 0)$, $B = (0, 3000)$ et $C = (0, 7000)$, si le robot se trouve initialement en $P = (0, 2000)$, il va aller en d'abord en B puis en A puis en C et parcourir $1 + 3 + 7 = 11$ mètres alors que le chemin $PABC$ de longueur $2 + 3 + 4 = 9$ mètres est plus court.

3 Deuxième approche : algorithme génétique

On aura plusieurs fois besoin de la liste des p premiers entiers de 0 à $p - 1$.
On peut la créer par des raccourcis syntactiques :

```
[k for k in range(p)] ou list(range(p))
```

Voici une fonction plus universelle, qui sera utilisée

```
def init(p):
    l = [0]*p
    for i in range(p):
        l[i] = i
    return l
```

3.1 Initialisation et évaluation

Exercice 18 — III.A

```
def creer_population(m, d):
    n = len(d) - 1
    population = [0]*m
    points0 = init(n-1)
    for i in range(m):
        chem = random.sample(points0, n-1)
        long = longueur_chemin(chem, d)
        population[i] = (long, chem))
    return population
```

3.2 Sélection

Exercice 19 — III.B

```
def reduire(p):
    m = len(p)
    p.sort()
    for i in range(m//2):
        p.pop()
```

3.3 Mutation

Exercice 20 — III.C.1)

```
def muter_chemin(c):
    n = len(c)
    i, j = random.sample(init(n), 2)
    temp = c[i]
    c[i] = c[j]
    c[j] = temp
```

Exercice 21 — III.C.2)

```
def muter_population(p, proba, d):
    m = len(p)
    for i in range(m):
        if random.random() <= proba:
            long, chem = p[i]
            muter_chemin(chem)
            long = longueur_chemin(chem, d)
            p[i] = (long, chem)
```

3.4 Croisement

Exercice 22 — III.D.1)
On doit normaliser car les deux parties peuvent avoir des sommets en commun.

```
def croiser(c1, c2):
    n = len(c1)
    gauche = c1[ : n//2]
    droit = c2[n//2 : n]
    return normaliser_chemin(gauche+droit, n)
```

Exercice 23 — III.D.2)

```
def nouvelle_generation(p, d):
    m = len(p)
    for i in range(m):
        l1, c1 = p[i]
        if i == (m-1):
            l2, c2 = p[0]
        else:
            l2, c2 = p[i + 1]
        chem = croiser(c1, c2)
        long = longueur_chemin(chem, d)
        p.append((long, chem))
```

3.5 Algorithme complet

Exercice 24 — III.E.1)

```
def algo_genetique(PI, m, proba, g):
    d = calculer_distances(PI)
    p = creer_population(m, d)
    for k in range(g):
        reduire(p)
        nouvelle_generation(p, d)
        muter_population(p, proba, d)
    indice_min = 0
    for i in range(1, len(p)):
        if p[i][0] < p[indice_min][0]:
            indice_min = i
    return p[indice_min]
```

Exercice 25 — III.E.2)
Le résultat peut se dégrader car on peut muter un individu réalisant le minimum à un instant donné. Pour éviter ce problème, on peut décider de ne pas muter un individu réalisant le minimum (ce qui oblige à le calculer à chaque itération), ou bien de ne muter un individu que si le mutant est meilleur que l'individu lui-même.

Exercice 26 — III.E.3)
On peut décider de s'arrêter lorsque :

- On a calculé un certain nombre de générations, c'est ce qui est proposé ici.
  Avantage : simplicité de la condition d'arrêt.
  Inconvénient : on a aucune idée sur la précision du résultat.
- On a calculé un certain nombre de générations, mais on a gardé le minimum de toutes les générations. On a juste une légère amélioration du critère précédent.
- Le meilleur chemin évolue peu sur plusieurs générations.
  Avantage : facilité de l'écriture.
  Inconvénients : on peut attendre longtemps, possibilité de minimum local.