# Images de vagues et de structures

## 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 (figure 1).

Les programmes demandés sont à rédiger en langage `Python` 3. Si toutefois le candidat utilise une version antérieure de `Python`, il doit le préciser. Il n'est pas nécessaire d'avoir réussi à écrire le code d'une fonction pour pouvoir s'en servir dans une autre question. Les questions portant sur les bases de données sont à traiter en langage `SQL`.

## Notations mathématiques et physiques

On note $\overrightarrow{V}$ un vecteur de $\mathbb{R}^3$, et `V` la variable qui lui est associée en `Python`. Dans l'ensemble du sujet, les vecteurs sont représentés par une liste de trois flottants. Par exemple, un vecteur dont les coordonnées sont $x = 2$, $y = 3$ et $z = 1{,}5$ sera exprimé par :

**code Python**

```
1  V = [2., 3., 1.5]
```

Partout où cela est nécessaire, les variables sont considérées être exprimées dans les unités SI. Le repère $(O,\overrightarrow{e_x},\overrightarrow{e_y},\overrightarrow{e_z})$ 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. 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 2(a) 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 $AB = AD = AC = 1$. Sa représentation en `Python` est donnée dans le paragraphe I.b.

**Organisation du sujet** Ce sujet se compose de trois parties indépendantes.

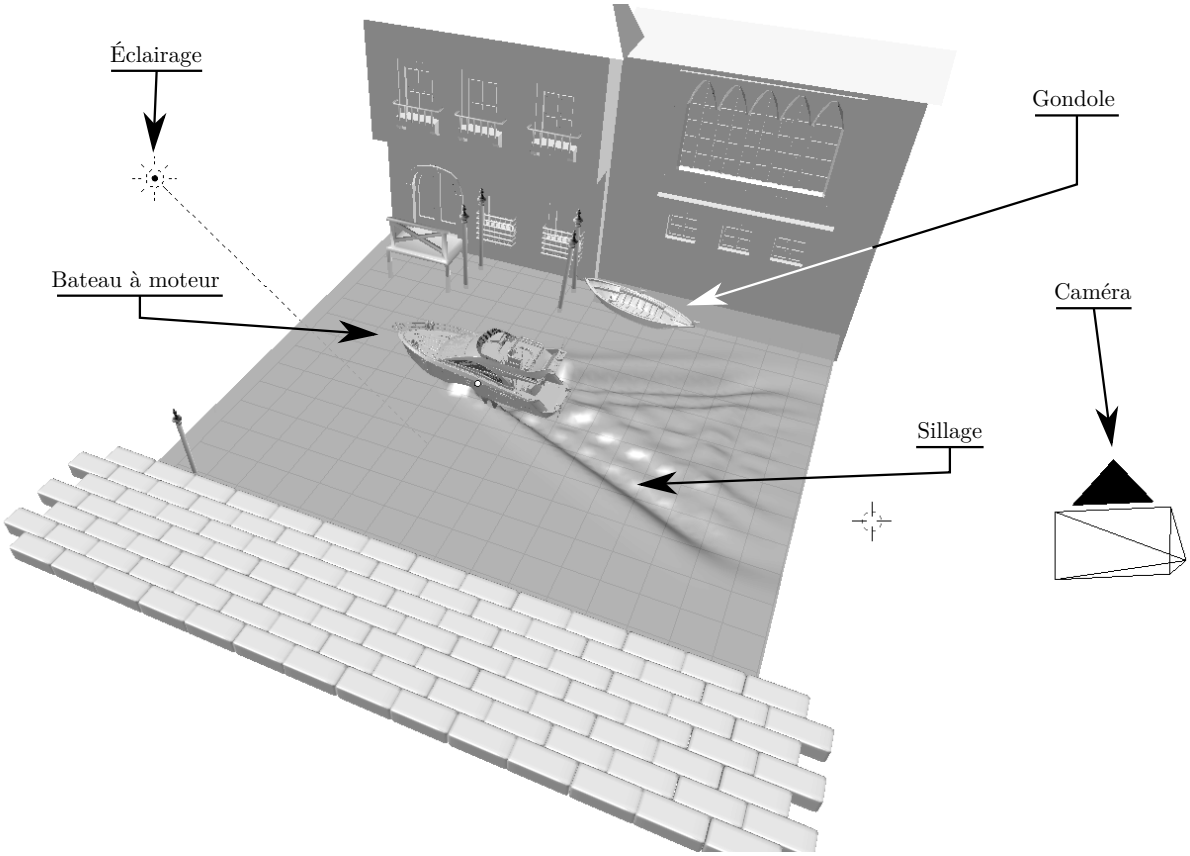
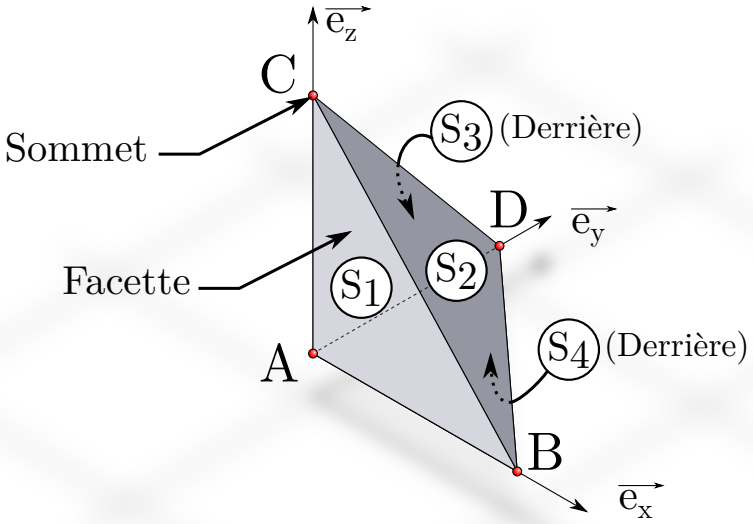
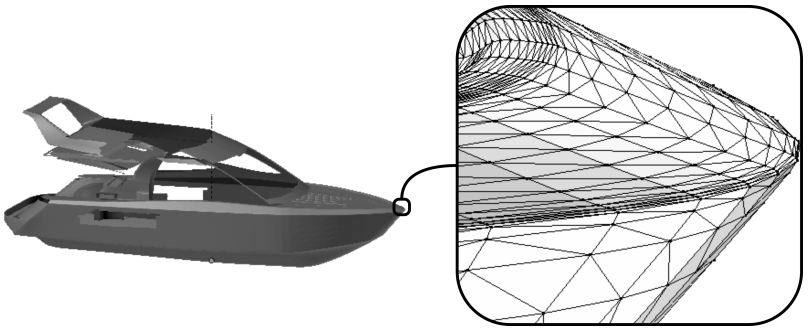



FIGURE 1 – Présentation de la scène étudiée.




(a) Un maillage simple : le tétraèdre



(b) Un maillage plus complexe : la coque d'un bateau.

FIGURE 2

## Partie I. Création d'un objet dans la scène

### I.a Chargement d'un modèle 3D à partir d'une base de données

On souhaite importer, dans `Python`, le modèle d'un bateau à moteur, élaboré préalablement. Ce modèle définit plusieurs maillages élémentaires (coque, bouée, échelle, moteur, etc.). Le fichier contenant ce modèle est une base de données relationnelle. Son schéma détaillé ci-dessous est illustré en figure 3 :

- relation `maillages_bateau` : ensemble des maillages. Un maillage possède un identifiant (entier) et un nom (chaîne de caractères) :

`maillages_bateau` (<u>`id`</u>,`nom`)

- relation `faces` : ensemble des facettes du modèle. Chaque facette est définie par un numéro unique, l'identifiant du maillage auquel elle appartient ainsi que les identifiants des sommets qui la composent. Tous sont des entiers.

`faces` (<u>`numero`</u>,`maillage`,`s1`,`s2`,`s3`)

- relation `sommets` : liste des sommets du modèle. Chaque sommet est défini par un identifiant (entier) et ses coordonnées dans l'espace par rapport au repère principal de la scène (flottant) :

`sommets` (<u>`id`</u>,`x`,`y`,`z`)

| <u>id</u> | nom |
|---|---|
| 1 | coque |
| 2 | bouée |
| 3 | échelle |
| 4 | moteur |
| ... | ... |

(a) Relation `maillages_bateau`

| <u>numero</u> | maillage | s1 | s2 | s3 |
|---|---|---|---|---|
| 1 | 3 | 1 | 2 | 3 |
| 2 | 3 | 2 | 4 | 3 |
| 3 | 2 | 3 | 12 | 5 |
| ... | ... | ... | ... | ... |

(b) Relation `faces`

| <u>id</u> | x | y | z |
|---|---|---|---|
| 1 | 0.0 | 0.0 | 0.0 |
| 2 | 1.0 | 0.0 | 0.0 |
| 3 | 0.0 | 1.0 | 0.0 |
| ... | ... | ... | ... |

(c) Relation `sommets`

FIGURE 3 – Exemples des relations de la base de données.

❑ **Q1** – ***Proposez une requête SQL permettant de compter le nombre de maillages que contient le modèle du bateau.***

**Réponse :**

**Requête SQL**

```
SELECT COUNT( * )
FROM maillages_bateau
```

❑ **Q2** – ***Proposez une requête SQL permettant de récupérer la liste des numéros des facettes (`numero`) du maillage nommé « gouvernail ».***

Réponse :

**Requête SQL**

```
1 SELECT f.numero
2 FROM maillages_bateau AS m JOIN faces AS f
3 ON m.id=f.maillage
4 WHERE m.nom="gouvernail"
```

❑ **Q3 – *Expliquez ce que renvoie la requête SQL suivante :***

**Requête SQL**

```
1 SELECT (MAX(x)-MIN(x))
2 FROM sommets AS s JOIN faces AS f JOIN  maillages_bateau AS m
3 ON (s.id=f.s1 OR s.id=f.s2 OR s.id=f.s3) AND f.maillage=m.id
4 WHERE m.nom="coque"
```

Réponse :

La requête renvoie la largeur du maillage nommé « coque » sur la direction $\vec{x}$.

### I.b Travail sur les facettes

À partir de requêtes sur la base de données précédente, on suppose avoir construit la variable `Python` suivante :

```
maillage_tetra = [ [[0.,0.,0.], [0.,0.,1.], [0.,1.,0.]],
                   [[0.,0.,0.], [0.,1.,0.], [1.,0.,0.]],
                   [[0.,0.,0.], [1.,0.,0.], [0.,0.,1.]],
                   [[1.,0.,0.], [0.,1.,0.], [0.,0.,1.]] ]
```

Cette variable représente un maillage. L'exemple illustré en figure 2(a) représente le tétraèdre (le point $A$ a pour coordonnées (0,0,0)).

❑ **Q4 – *À partir de la variable* `maillage_tetra`*, écrire une expression* `Python` *permettant de récupérer la coordonnée $y$ du premier sommet de la première facette.***

Réponse :

**code Python**

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

❑ **Q5 – *À quel élément, sur la figure 2(a), correspond* `maillage_tetra[1]` *?**

**Réponse :**

`maillage_tetra[1]` se réfère à la face dont les sommets sont :

- [0.,0.,0.]= A
- [0.,1.,0.]= D
- [1.,0.,0.]= B

Il s'agit donc de la face $S_4$.

Dans le reste du sujet, on suppose qu'on dispose d'un module `Python` dont la documentation suit.

Help on module operations_vectorielles :

**NAME**

operations_vectorielles

**DESCRIPTION**

Ce module contient des fonctions qui implémentent des opérations usuelles sur les vecteurs. Sauf indication contraire explicite les arguments sont des vecteurs passés sous la forme de liste de trois réels.

**FUNCTIONS**

**addition**(V1, V2)

Renvoie le vecteur correspondant à l'opération vectorielle V1+V2.

**aire**(F)

Renvoie l'aire d'une facette.
Argument :
F – une facette (liste de trois vecteurs)

**prod_scalaire**(V1, V2)

Renvoie le produit scalaire de V1 avec V2.

**prod_vectoriel**(V1, V2)

Renvoie le vecteur correspondant au produit vectoriel de V1 avec V2.

**soustraction**(V1, V2)

Renvoie le vecteur correspondant à l'opération vectorielle V1-V2.

**barycentre**(F)

Renvoie le vecteur position du barycentre d'une facette.
Argument :
F – une facette (liste de trois vecteurs)

**FILE**

operations_vectorielles.py

❑ **Q6** – ***On souhaite utiliser les fonctions de ce module depuis un autre fichier*** `Python`***. Complétez le code ci-dessous afin qu'il fournisse le résultat attendu :***

**code Python**

```
1 from ??? import ?? as ?
2 vect_1 = [1., 2., 3.]
3 vect_2 = [2., 3., 4.]
4 scal12 = ps(vect_1, vect_2) #Calcul du produit scalaire de vect_1 avec vect_2
```

**Réponse :**

**code Python**

```
1 from operations_vectorielles import prod_scalaire as ps
```

Dans ce module, une fonction n'a pas été documentée :

**code Python**

```
1 def mystere1(V):
2     return (V[0]**2 + V[1]**2 + V[2]**2)**0.5
```

❑ **Q7** – ***Que fait la fonction*** `mystere1` ***?***

**Réponse :**

Elle calcule la norme Euclidienne d'un vecteur V.

❑ **Q8** – ***Créer la fonction*** `multiplie_scalaire`***, prenant comme argument un flottant*** `a` ***et un vecteur*** `V` ***et renvoyant un nouveau vecteur correspondant à*** $a\,\vec{V}$ ***.***

**Réponse :**

**code Python**

```
1 def multiplie_scalaire(a, v):
2     return [a*k for k in v]
```

La fonction `barycentre` (incomplète) est définie ci-dessous.

**code Python**

```
1 def barycentre(F):
2     G = [0,0,0]
3     for i in range(3):  #Pour chaque point de F
4         .....    # Ligne à compléter
5         .....    # Ligne à compléter
6     return G
```

❑ **Q9** – ***Compléter les lignes 4 et 5 permettant de calculer le barycentre.***

Réponse :

**code Python**

```
1 |   |   for k in range(3):  #Pour chaque coordonnées
2 |   |   |   G[k] += F[i][k] / 3.
```

❑ **Q10** – ***Pour une facette*** `F=(A,B,C)` ***d'aire non-nulle, proposer une fonction*** `normale`***, prenant comme argument une facette*** `F` ***et renvoyant le vecteur unitaire normal***

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

Réponse :

**code Python**

```
1 def normale(F):
2         AB = soustraction(f[1], f[0])
3         AC = soustraction(f[2], f[0])
4         V = prod_vectoriel(AB, AC)
5         n = multiplie_scalaire(1/mystere1(V), V) # mystere1 = norme, pour
  rappel
6         return n
```

### I.c Liste des sommets

❑ **Q11** – ***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. Proposer une fonction*** `sont_proches`***, prenant comme arguments deux sommets*** `S1` ***et*** `S2` ***(représentés par leur vecteur position) et un flottant positif*** `eps`***, et qui renvoie*** `True` ***si*** `S1` ***et*** `S2` ***sont proches (i.e. si leur distance au sens de la norme Euclidienne est inférieure à*** `eps`***) et*** `False` ***sinon.***

Réponse :

**code Python**

```
1 def sont_proches(S1, S2, eps):
2     return mystere1(soustraction(S1,S2)) < eps
```

Soient les fonctions suivantes :

**code Python**

```
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
```

❑ **Q12** – ***Sous quelle condition la fonction*** `mystere2` ***renvoie-t-elle*** `True` ***?***

**Réponse :**

`mystere2` renvoie `True` si le point `S1` est proche d'un point de la liste `L`.

❑ **Q13** – ***Donner (sans justification) ce que renvoie*** `mystere3(maillage_tetra)`, ***dans le cas où*** `maillage_tetra` ***est la variable définie précédemment.***

**Réponse :**

On pourra noter que le `return res` est mal placé (la syntaxe est correcte mais la fonction ne produit pas l'effet supposé). En l'état la fonction retourne

```
[ [0.,0.,0.], [0.,0.,1.], [0.,1.,0.] ]
```

alors que s'il était bien placé, la fonction retournerait

```
[ [0.,0.,0.], [0.,0.,1.], [0.,1.,0.], [1.,0.,0.] ]
```

On acceptera les deux réponses et on pourra considérer que les candidats ayant vu le mauvais placement du `return res` seront de bons candidats.

❑ **Q14** – ***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.***

**Réponse :**

Si les candidats ont compris que le `return res` est mal placé alors ils vont pouvoir répondre assez facilement à la question. En effet, en ce cas la fonction `mystere3` à une complexité constante.
S'ils ne l'ont pas vu, ils doivent répondre comme-suit.

**Complexité de `mystere2` :**

- **Meilleur des cas :** ■ `P1` ■ est présent dès le début de ■ `L` ■. La boucle `for` n'est exécutée qu'une seule fois. La complexité est $\boxed{\text{un } \mathcal{O}(1)}$
- **Pire des cas :** ■ `P1` ■ n'est pas présent dans ■ `L` ■. Le test avec `sont_proche` est toujours faux et la boucle `for` est exécutée $n$ fois. La complexité est $\boxed{\text{un } \mathcal{O}(n)}$

**Complexité de `mystere3` :**

La première boucle `for` est exécutée $m$ fois. La seconde boucle `for` est exécutée 3 fois (3 points par facette).

- **Meilleur des cas :** Si `mystere2` est un $\mathcal{O}(1)$ (tous les points sont au même endroit), la complexité est alors $\mathcal{O}(3 \times m \times 1) = \boxed{\mathcal{O}(m)}$.
- **Pire des cas :** Aucun point en commun d'une facette à l'autre (chaque point est unique) : `mystere2` est toujours dans le pire des cas $\mathcal{O}(n)$, avec $n = 1$ puis 2 puis 3, ... ,puis $(3 \times m)$, au fur et à mesure que `res` accumule les points. La complexité est alors $\mathcal{O}\left(\frac{(3m)(3m+1)}{2}\right) = \boxed{\mathcal{O}(m^2)}$

## Partie II. Génération de vagues

Le bateau à moteur qui traverse le canal génère des vagues (appelées *sillage de Kelvin*). On ne dispose de formules analytiques décrivant ces ondes que dans des cas très simples, comme celui d'un bateau en translation rectiligne uniforme.
Le calcul numérique permettant d'évaluer la forme de ces vagues étant coûteux, il est réalisé par un programme extérieur. Ce dernier génère l'état du plan d'eau à chaque image de la scène (25 par seconde).
Pour chacune de ces images, on représente le plan d'eau par une grille régulière carrée, de taille $(m+1) \times (m+1)$ (figure 4). Chaque case $(i,j)$ de cette grille correspond à un sommet du maillage de la surface de l'eau, noté $n_{i,j}$ de coordonnées $(x_i,y_j)$ (avec $(i,j) \in [\![0,m]\!]^2$).
La hauteur (sur $\overrightarrow{e_z}$) de $n_{i,j}$ est notée $h_{i,j}$. L'ensemble de tous les $h_{i,j}$ est stocké dans une liste de listes nommée `mat_h`, tel que `mat_h[i][j]`$= h_{i,j}$.
Le programme extérieur calcule ainsi `mat_h` pour chaque nouvelle image. L'ensemble de toutes les valeurs de `mat_h` est stocké dans une liste nommée `liste_vagues`.
La scène est composée de 350 images. Le plan d'eau est composé de $200 \times 200$ sommets. Chaque hauteur $h_{i,j}$ est un flottant codé sur 64 bits.

❑ **Q15 –** ***Quel est l'espace occupé en mémoire vive par l'ensemble des données (en Mo).***

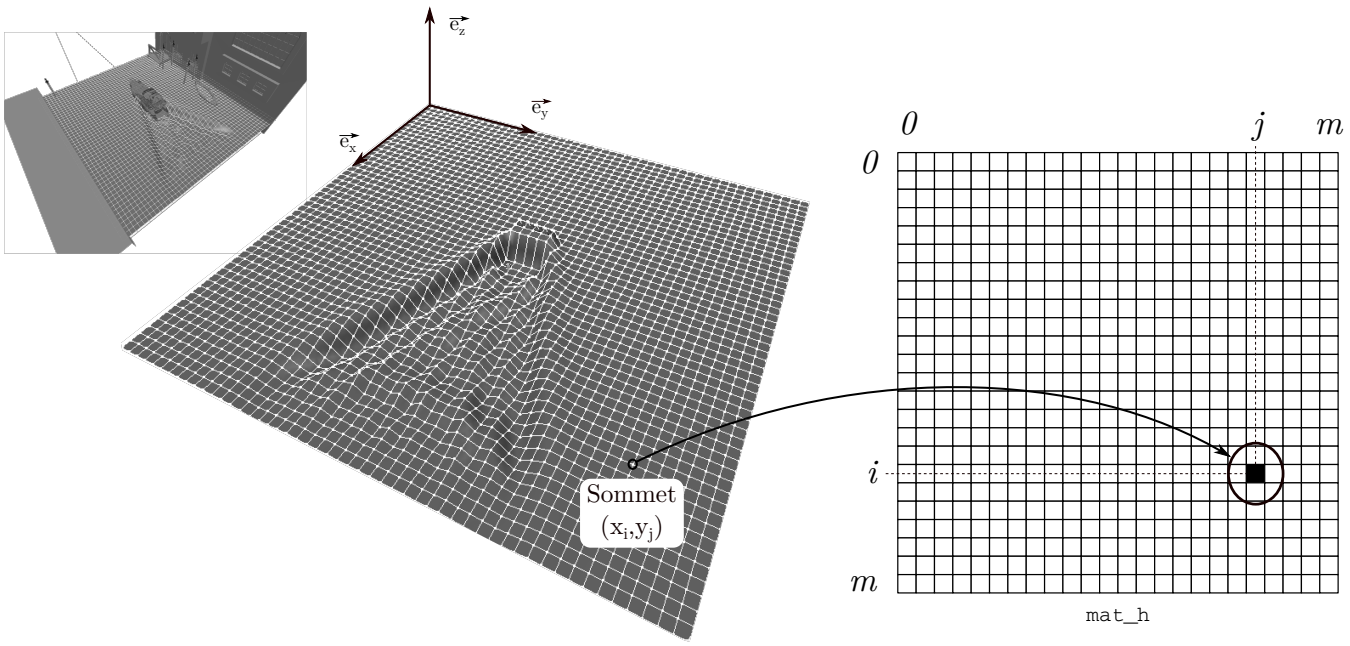



FIGURE 4 – Illustration du stockage de la hauteur d'eau dans un tableau.

**Réponse :**

$$\frac{64}{8} \times 200 \times 200 \times 350 = 112 \cdot 10^6\, o$$
$$= 112\, Mo$$

On souhaite enregistrer le contenu de `liste_vagues` dans un fichier afin de le transmettre au logiciel d'animation.

❑ **Q16** – ***Écrire une fonction*** `mat2str` ***qui prend en argument une liste de listes (représentant un*** `mat_h`***) et renvoie les données qu'elle contient sous forme d'une chaîne de caractères qui respecte le format suivant :***

$$\begin{array}{|l} h_{01};h_{02};\ \ldots\ ;h_{0m} \\ h_{11};\ \ldots\ \ \ ;h_{1m} \\ \ldots \\ h_{m0};\ \ldots\ \ \ ;h_{mm} \end{array}$$

***On rappelle que le retour à la ligne est codé par le caractère "\n".***

Réponse :

**code Python**

```
def mat2str(mat):
    resultat = ""
    tag_premiere_ligne = True
    for i in range(len(mat)):  # Pour chaque cellule d'une matrice
        if tag_premiere_ligne:  # Gestion des sauts de ligne
            tag_premiere_ligne = False
        else:
            resultat += "\n"
        tag_premiere_colonne = True
        for j in range(len(mat[i])):
            if tag_premiere_colonne: #Si c'est pas la 1ère colonne...
                tag_premiere_colonne = False
            else :
                resultat += ";"
            resultat += str(mat[i][j])
    return resultat
```

❑ **Q17** – ***En s'appuyant sur*** `mat2str`***, proposer un code*** `Python` ***qui permet de sauvegarder le contenu de*** `liste_vagues` ***dans un fichier nommé*** ■ `fichier_vagues.txt` ■ ***(dans le répertoire courant), en séparant la représentation de chaque*** `mat_h` ***par deux sauts de lignes consécutifs.***

Réponse :

**code Python**

```
fichier = open("fichier_vagues.txt",'w')
tag_premiere_matrice = True
for mat in liste_vagues:
    if tag_premiere_matrice:  # Gestion des sauts de ligne
        tag_premiere_matrice = False
    else:
        fichier.write("\n\n")
    fichier.write(mat2str(mat))
fichier.close()
```

Le fichier texte obtenu est jugé trop volumineux. On décide de recourir à des matrices creuses. Dans ce qui suit on désigne par matrice creuse une matrice dont seuls la valeur et l'emplacement des éléments non-nuls (c'est-à-dire significativement éloignés de zéro) sont enregistrés.

On propose d'utiliser le format de fichier nommé « Coordinate Format » qui consiste à stocker :

- une liste `I`, comportant les numéros de ligne de chaque élément non-nul,
- une liste `J`, comportant les numéros de colonne de chaque élément non-nul,
- une liste `N`, comportant la valeur de chaque élément non-nul.

Les valeurs d'une liste sont enregistrées sur une même ligne et séparées par des points-virgules. Chaque liste est séparée des autres par un retour à la ligne. On admet que les flottants sont écrits dans le fichier avec 15 caractères (tout compris).

❑ **Q18** – ***Après avoir défini judicieusement les types des éléments contenus dans*** `I, J` ***puis*** `N`***, estimer la taille (en octets) que prendra une matrice ayant*** $p$ ***éléments non-nuls, au format « Coordinate Format », dans le fichier.***

**Réponse :**

- `I` et `J` contiennent des entiers compris entre 0 et 200 (3 caractères max pour chaque).
- `N` contient des flottants (15 caractères pour chaque).
- il y aura $(p-1)$ points-virgules et 2 retours à la ligne

Chaque matrice fera donc $\underbrace{(3+3+15)\times p}_{\text{Place des nombres}} + \underbrace{3*(p-1)}_{\text{";"}} + \underbrace{2}_{\text{"}\backslash n\text{"}} = \boxed{(21p-1) \text{ octets}}$. On acceptera les réponses où un entier est représenté par un seul octet.

❑ **Q19** – ***En déduire à partir de combien d'éléments non-nuls il devient moins avantageux d'enregistrer une matrice creuse qu'une matrice complète classique.***

**Réponse :**

Une matrice complète contenant $p$ valeurs "significativement" non-nulles prendrait :

- 15 caractère par nombre non-nul (il y en a $p$),
- 1 caractère pour chaque "0" (il y en a $(200^2-p)$),
- 1 caractère à droite de chaque nombre (soit un ";", soit un saut de ligne – Il y en a $200^2$.)

$$p\times 15 + (200^2-p)\times 1 + 200^2\times 1 = 14p + 80000$$

$$\begin{aligned} &\text{Taille matrice creuse} > \text{Taille matrice pleine} \\ &\Leftrightarrow (21p-1) > 14p + 80000 \\ \Leftrightarrow p > \frac{80000+1}{7} &\approx \boxed{11400 \text{ cellules}} \end{aligned}$$

❑ **Q20** – ***Proposer un code permettant de construire, pour un tableau*** `mat_h` ***donné, les listes*** `Python I,J` ***et*** `N`***. On considérera nulles les hauteurs inférieures à*** $10^{-3}$ ***(en valeur absolue).***

**Réponse :**

**code Python**

```
I = []
J = []
N = []
for i in range(len(mat_h)):
    for j in range(len(mat_h[i])):
        if abs(mat_h[i][j]) >= 10**(-3):
            I.append(i)
            J.append(j)
            N.append(mat_h[i][j])
```

## Partie III. Mouvement de flottaison

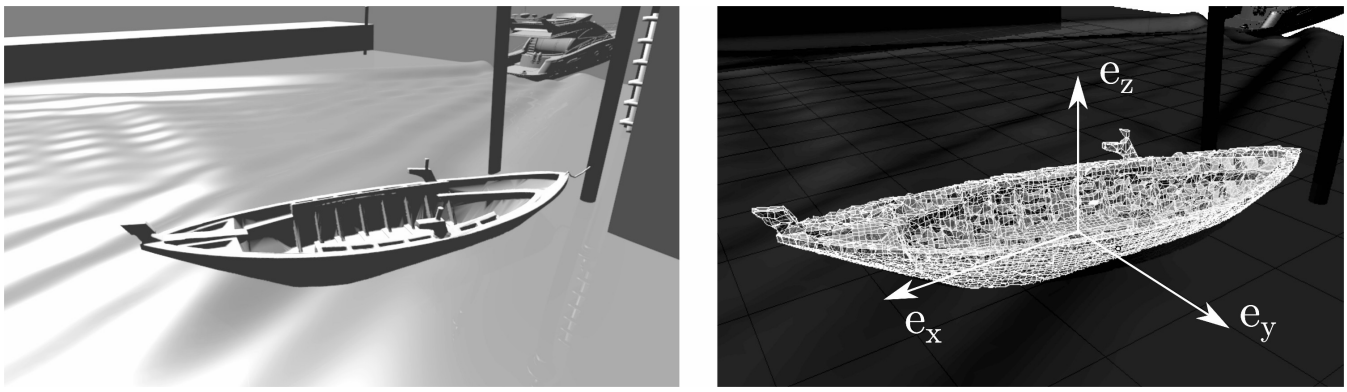



FIGURE 5 – Maillage de la gondole.

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.

### III.a Estimation de la poussée d'Archimède

Seul le mouvement de translation verticale (selon la direction verticale $\overrightarrow{e_z}$) 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 facette nommée `maillageG`, similaire à celle présentée dans la partie précédente.

❑ **Q21 –** ***Proposer une fonction*** `lister_FI` ***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 partie I.***

**Réponse :**

**code Python**

```
def lister_FI(M):
    liste_immergees = []
    for facette in M:
        centre = barycentre(facette)
        if centre[2] <= hauteur(centre[0], centre[1]):
            liste_immergees.append(facette)
    return liste_immergees
```

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 :

$$\overrightarrow{F_i} = -S_i \times p(G_i)\,\overrightarrow{n_i}$$

avec :
- $S_i$ : l'aire de la facette,
- $\overrightarrow{n_i}$ : le vecteur normal sortant de la coque,
- $p(G_i)$ : 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 :

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

avec :
- $\rho$ : masse volumique de l'eau ($\rho \approx 1000$)
- $g$ : accélération de la pesanteur (ici : $g \approx 9{,}81$)

❑ **Q22 –** ***Proposer une fonction*** `force_facette` ***prenant en argument une facette*** `F`***, et renvoyant le vecteur force appliqué par l'eau sur cette facette. On pourra utiliser les fonctions définies précédemment.***

Réponse :

**code Python**

```
def pression(point):
    rho = 1000
    g = 9.81
    return rho * g * (hauteur(point[0],point[1]) - point[2])

def force_facette(F):
    surface = aire(F)
    vectDirecteur = normale(F)
    cdg = barycentre(F)
    p = pression(cdg)
    return multiplie(surface*pression, vectDirecteur)
```

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

❑ **Q23** – ***Définir la fonction*** `resultante` ***prenant comme argument une liste*** `L` ***de facettes (supposées immergées), renvoyant la somme des forces sur l'axe*** $\vec{z}$ ***de l'eau, appliquée sur l'ensemble des surfaces.***

Réponse :

**code Python**

```
def resultante(L):
    res = 0
    for facette in L:
        res += forceFacette(facette)[2]
    return res
```

### III.b Tri des facettes

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 souhaite donc 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). On propose le code (incomplet) du tri-fusion ci-dessous :

**code Python**

```
def fusion(L1, L2):
    # À compléter (sur une ou plusieurs lignes)

def trier_facettes(L):
    # À compléter (sur une ou plusieurs lignes)

grandesFacettes = # À compléter
```

❑ **Q24** – ***Compléter la fonction*** `fusion`***, prenant comme argument deux listes de facettes*** `L1` ***et*** `L2` ***(supposées triées par aire décroissante) et renvoyant une nouvelle liste composée des facettes de*** `L1` ***et*** `L2` ***triées par aire décroissante.***

**Réponse :**

Réponse 1 :

**code Python**

```
def fusion(L1,L2):
    n1, n2 = len(L1), len(L2)
    liste_face = []
    i, j = 0, 0
    while i < n1 and j < n2 :
        if aire(L1[i]) > aire(L2[j]):
            liste_face.append(L2[j])
            j += 1
        else :
            liste_face.append(L1[i])
            i += 1
    if i == n1 :
        liste_face.extend(L2[j:])
    else :
        liste_face.extend(L1[i:])
    return liste_face
```

Réponse 2 :

**code Python**

```
def fusion(L1, L2):
   if len(L1) == 0:
       return L2
   if len(L2) == 0:
       return L1

   if aire(L1[0]) > aire(L2[0]):
       return [L1[0]]+fusion(L1[1:],L2)
   else:
       return [L2[0]]+fusion(L1,L2[1:])
```

❑ **Q25** – ***Compléter la fonction récursive*** `trier_facettes`***, prenant comme argument une liste de facettes*** `L`***, et renvoyant une nouvelle liste de facettes triées dans l'ordre des aires décroissantes, par la méthode du tri-fusion.***

Réponse :

**code Python**

```
def trier_facettes(L):
    n = len(L)
    if n<= 1:
        return L

    iPivot = n//2

    L1 = trier_facettes(L[0:iPivot])
    L2 = trier_facettes(L[iPivot:n])

    return fusion(L1, L2)
```

❑ **Q26** – ***Affecter à une nouvelle variable*** `grandesFacettes` ***la liste des facettes de*** `maillageG`***, privée de la moitié des facettes les plus petites (en cas de nombre impair d'éléments, on inclura la facette médiane).***

Réponse :

**code Python**

```
facettes_triees = trier_facette(maillageG)
gF = facettes_triees[0:(len(maillageG)+1)//2]
```

### III.c Mouvement vertical de la gondole

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 $\overrightarrow{e_z}$, appliqué à la gondole énonce que :

$$\frac{dv}{dt} = \frac{1}{m} \times F_{eau \mapsto gondole} - g$$
$$\frac{dz_G}{dt} = v$$

avec

- $m$ : masse de la gondole
- $F_{eau \mapsto gondole}$ 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 $z_{G0} = 0$. 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 (liste de trois flottants) ;

- `vitG` contiendra le vecteur vitesse actuel de ce même point au moment de l'appel (liste de trois flottants) ;
- `mailG` contiendra la liste des grandes facettes de la gondole (privée des petites, au sens de la question précédente), au moment de l'appel.

**code Python**

```
1 def nouvelle_hauteur(posG, vitG, mailG):
2     dt=1.0/25.0 # Pas de temps correspondant à une image du film.
3     facettes_immergees = lister_FI(mailG)
4     posG = posG + ..........    # à compléter
5     vitG = vitG + ...........   # à compléter
6     return posG, vitG
```

❑ **Q27 – *Compléter les lignes 4 et 5 du code précédent conformément à la méthode d'Euler.***

**Réponse :**

L'énonce stipule qu'on utilise des listes (et non des tableaux `numpy`). Les lignes 4 et 5 procèdent donc à des extensions de listes, si bien que la fonction retourne 2 listes de 6 éléments. On suppose que la masse `m` est une variable globale.

Ligne 4 :

**code Python**

```
posG = posG + [posG[0]+vitG[0]*dt,
               posG[1]+vitG[1]*dt,
               posG[2]+vitG[2]*dt]
```

Ligne 5 :

**code Python**

```
vitG = vitG + [vitG[0]+0.0,
               vitG[1]+0.0,
               vitG[2]+( resultante(facettes_immergees)/m -9.81 )*dt]
```

FIN DE L'ÉPREUVE