Corrigé du sujet d'informatique commune Mines-Ponts 2020

Arnaud MONCET, Lycée de la Borde Basse

Q1.
```
SELECT COUNT(*) FROM maillages_bateau
```

Q2.
```
SELECT numero
    FROM faces JOIN maillages_bateau ON maillage=id
    WHERE nom="gouvernail"
```

Q3. La requête renvoie l'écart maximal entre 2 abscisses dans le maillage nommé « coque ».

Q4. `y = maillage_tetra[0][0][1]`

Q5. `maillage_tetra[1]` correspond à la facette S_4.

Q6. Ligne 1 : `from operations_vectorielles import prod_scalaire as ps`

Q7. La fonction `mystere1` renvoie la norme du vecteur `V` passé en argument.

Q8.
```
def multiplie_scalaire(a,V):
    return [a*V[0], a*V[1], a*V[2]]
```

Q9.
```
for j in range(3):
    G[j] += F[i][j]/3
```

Q10. Pour cette question et les suivantes, on suppose que la bibliothèque `operations_vectorielles` a été importée en écrivant `from operations_vectorielles import *` .

```
def normale(F):
    A, B, C = F
    AB = soustraction(B, A)
    AC = soustraction(C, A)
    V = prod_vectoriel(AB, AC)
    return multiplie_scalaire(1/mystere1(V), V)
```

Q11.
```
def sont_proches(S1, S2, eps):
    return mystere1(soustraction(S1, S2)) <= eps
```

Q12. 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}.

Q13. `mystere3(maillage_tetra)` renvoie `[ [0.,0.,0.], [0.,0.,1.], [0.,1.,0.], [1.,0.,0.] ]`.

Q14.
- Complexité de `mystere2` (pour une liste `L` de longueur n) :
  Elle est proportionnelle au nombre de passages dans la boucle car chaque itération se fait en $O(1)$ (complexité constante pour la fonction `sont_proches`) et le reste est négligeable.
  Le meilleur cas est celui où `S1` est proche du premier élément de `L`. On a alors un seul passage dans la boucle, d'où une complexité en $O(1)$.
  Le pire cas est celui où `S1` n'est proche d'aucun élément de `L` ou est proche seulement du dernier. On a alors n passages dans la boucle, d'où une complexité en $O(n)$.

- Complexité de `mystere3` (pour un maillage contenant m facettes triangulaires) :
  Dans le meilleur cas, tous les sommets sont identiques donc la complexité du test ligne 12 est toujours en $O(1)$. Ce test est répété $3m$ fois (car il y a 3 sommets par facette), la ligne 13 n'est exécutée qu'une seule fois lors du premier passage dans les deux boucles. Finalement, la complexité dans le meilleur cas est $O(m)$.
  Dans le pire cas, aucun sommet n'est proche d'un autre sommet. La liste `res` a une longueur qui augmente de 1 à chaque itération, et la complexité pour chaque itération est celle de `mystere2` dans le pire cas, c'est-à-dire de l'ordre de la longueur de `res`. Il y a au total $3m$ itérations, donc la complexité de l'algorithme est de l'ordre de $1 + 2 + \cdots + 3m = \dfrac{3m(3m+1)}{2}$, soit $O(m^2)$.

Q15. L'espace occupé en mémoire est : $\dfrac{350 \times 200 \times 200 \times 64}{8 \times 10^6} = 112$ Mo.

Q16.
```
def mat2str(mat_h):
    ch = ""
    for i in range(len(mat_h)):
        for j in range(len(mat_h[i])):
            ch += str(mat_h[i][j])
            if j != len(mat_h[i])-1 :
                ch += ";"
            else :
                ch += "\n"
    return ch
```

Q17.
```
fichier = open("fichiers_vagues.txt", "w")
for mat_h in liste_vagues:
    fichier.write(mat2str(mat_h)+"\n\n")
fichier.close()
```

Q18. Les éléments de `I` et `J` sont des entiers, ceux de `N` sont des flottants.
On estime qu'en moyenne, les entiers compris entre 0 et $m = 199$ sont écrits sur $2,5$ caractères.
Ainsi, le nombre de caractères pour chacune des listes `I` et `J` est environ $3,5\,p$ (en comptant les points-virgules et le retour à la ligne à la fin).
Le nombre de caractères pour la liste `N` est quant à lui égal à $16\,p$.
Le nombre total de caractères pour représenter la matrice est donc environ $23\,p$.
En supposant que chaque caractère est codé sur 1 octet (ce qui est par exemple le cas pour le codage ASCII), la taille que prendra la matrice dans le fichier est donc de $23\,p$ octets (environ).

Q19. Pour écrire une matrice complète classique de taille 200×200, on a besoin de $200 \times 200 \times 15$ caractères pour écrire tous les flottants, puis 200×200 autres caractères pour les points-virgules et les retours à la ligne, ce qui fait au total $200 \times 200 \times 16 = 640\,000$ caractères. L'espace occupé en mémoire est donc égal à $200 \times 200 \times 16 = 640\,000$ octets.
$23\,p \geq 640\,000 \iff p \geq \dfrac{640\,000}{23} \approx 28\,000$: à partir de 28 000 éléments non nuls environ (sur un total de $(m+1)^2 = 40\,000$ éléments dans la matrice, soit 70 % d'éléments non nuls), il devient moins avantageux d'enregistrer une matrice creuse qu'une matrice classique.

Q20.
```
I, J, N = [], [], []
for i in range(len(mat_h)):
    for j in range(len(mat_h[i])):
        if abs(mat_h[i][j]) > 1e-3:
            I.append(i)
            J.append(j)
            N.append(mat_h[i][j])
```

Q21.
```
def lister_FI(M):
    L = []
    for facette in M:
        x, y, z = barycentre(facette)
        if z < hauteur(x,y):
            L.append(facette)
    return L
```

Q22. On suppose que des variables globales `rho` et `g` ont été définies.
On suppose aussi que les sommets d'une facette sont rentrés dans le bon ordre pour que le vecteur normal calculé par la fonction `normale` soit bien sortant de la coque (c'est le cas pour l'exemple `maillage_tetra` de la page 5 de l'énoncé).

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

Q23.
```
def resultante(L):
    res = 0.
    for F in L:
        res += force_facette(F)[2]
    return res
```

Q24.
```
def fusion(L1, L2):
    L = []
    while len(L1) != 0 and len(L2) != 0:
        if aire(L1[0]) > aire(L2[0]):
            L.append(L1.pop(0))
        else :
            L.append(L2.pop(0))
    if len(L1) == 0:
        L.extend(L2)
    else:
        L.extend(L1)
    return L
```

Q25.
```
def trier_facettes(L):
    if len(L) <= 1:
        return L
    else:
        m = len(L)//2
        L1 = trier_facettes(L[:m])
        L2 = trier_facettes(L[m:])
        return fusion(L1, L2)
```

Q26. `grandesFacettes = trier_facettes(maillageG)[:(len(maillageG)+1)//2]`

Q27. On suppose que des variables globales `m` et `g` ont été définies.
Ligne 4 : `posG = posG + dt*vitG`
Ligne 5 : `vitG = vitG + dt*(1/m*resultante(facettes_immergees)-g)`