

# Rapport TP3 – Quaternions

## Sommaire

Question 1 — Structure de corps des quaternions .....	2
1.a – Affichage conventionnel d'un quaternion.....	2
1.b – Somme de deux quaternions .....	3
1.c – Produit de deux quaternions .....	4
1.d – Conjugué, module et inverse d'un quaternion .....	5
Question 2 – Forme polaire.....	6
2.a – Forme polaire d'un quaternion .....	6
2.b – Conversion polaire → algébrique.....	7
Question 3 – Vérification de l'associativité .....	8
Question 4 – Quaternions et rotations .....	9
4.a – Matrice de rotation à partir d'un vecteur unitaire et d'un angle.....	9
4.b – Matrice de rotation à partir d'un quaternion unitaire.....	10
4.c – Retrouver un quaternion unitaire à partir d'une matrice de rotation .....	11

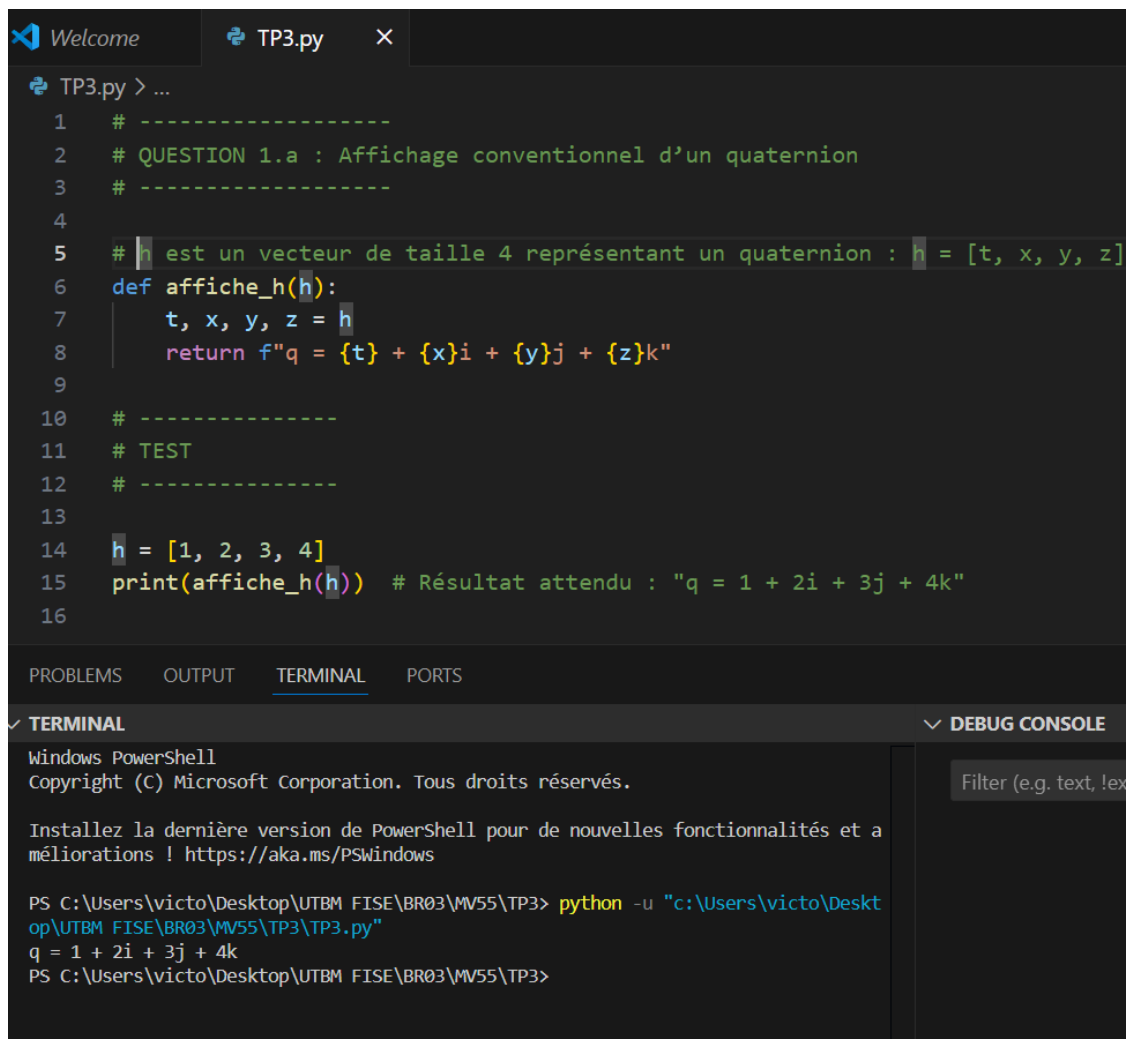
# Question 1 — Structure de corps des quaternions

## 1.a – Affichage conventionnel d'un quaternion

Dans cette première question, on considère un quaternion  $q = t + xi + yj + zk$ , que l'on représente sous forme de vecteur  $h = (t, x, y, z)$ . L'objectif est d'écrire une fonction Python qui renvoie une chaîne de caractères représentant ce quaternion dans sa forme algébrique usuelle.

La fonction `affiche_h(h)` prend en entrée un vecteur  $h \in \mathbb{R}^4$ , et renvoie la chaîne :  $q = t + xi + yj + zk$ , où les valeurs numériques sont remplacées par les composantes du vecteur.

Un test effectué sur le vecteur  $h = [1, 2, 3, 4]$  permet de vérifier que l'affichage est correct.



```
TP3.py > ...
1 # -----
2 # QUESTION 1.a : Affichage conventionnel d'un quaternion
3 # -----
4
5 # h est un vecteur de taille 4 représentant un quaternion : h = [t, x, y, z]
6 def affiche_h(h):
7     t, x, y, z = h
8     return f"q = {t} + {x}i + {y}j + {z}k"
9
10 # -----
11 # TEST
12 # -----
13
14 h = [1, 2, 3, 4]
15 print(affiche_h(h)) # Résultat attendu : "q = 1 + 2i + 3j + 4k"
16
```

PROBLEMS OUTPUT TERMINAL PORTS

✓ **TERMINAL** DEBUG CONSOLE

Windows PowerShell  
Copyright (C) Microsoft Corporation. Tous droits réservés.

Installez la dernière version de PowerShell pour de nouvelles fonctionnalités et améliorations ! <https://aka.ms/PSWindows>

PS C:\Users\victo\Desktop\UTBM FISE\BR03\MV55\TP3> python -u "c:\Users\victo\Desktop\UTBM FISE\BR03\MV55\TP3\TP3.py"

q = 1 + 2i + 3j + 4k

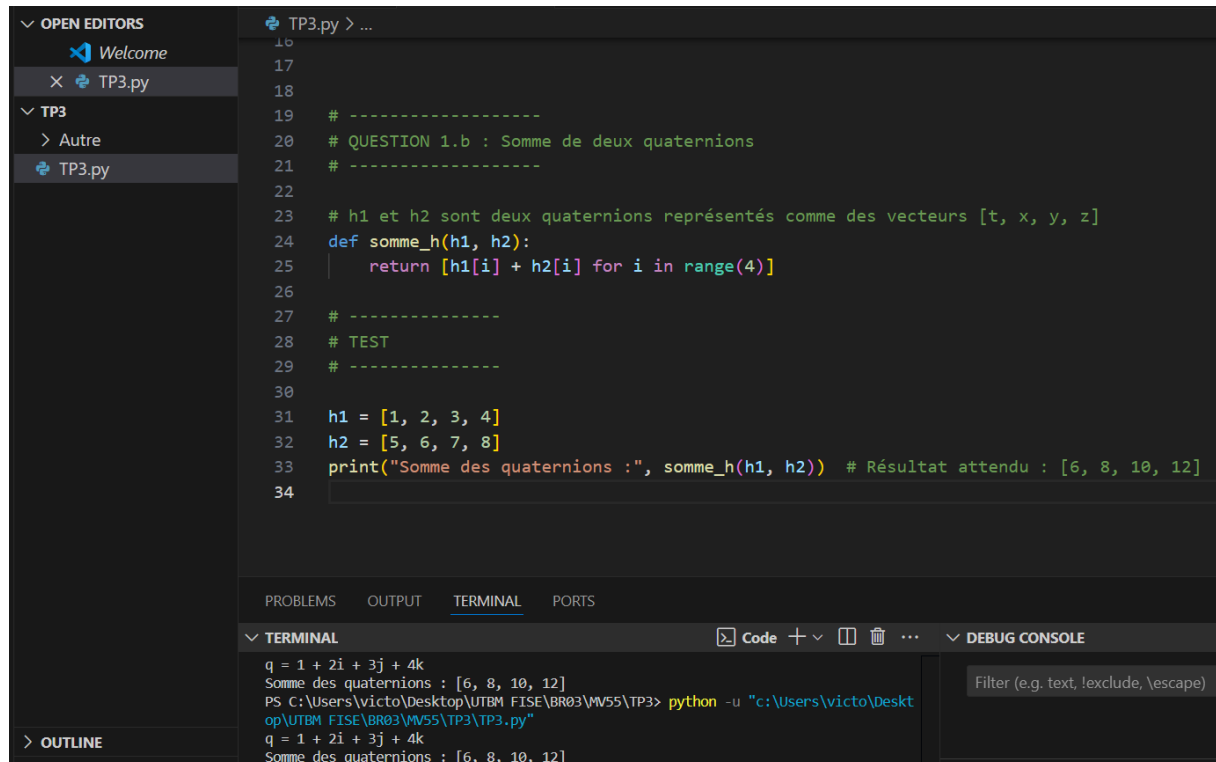
PS C:\Users\victo\Desktop\UTBM FISE\BR03\MV55\TP3>

## 1.b – Somme de deux quaternions

La somme de deux quaternions  $q_1 = t_1 + x_1i + y_1j + z_1k$  et  $q_2 = t_2 + x_2i + y_2j + z_2k$  correspond à l'addition terme à terme de leurs composantes.

Autrement dit, on additionne les parties scalaires et vectorielles séparément, ce qui revient à faire la somme de deux vecteurs de  $\mathbb{R}^4$ .

La fonction `somme_h(h1, h2)` prend en entrée deux quaternions sous forme de vecteurs, et retourne leur somme, également sous forme de vecteur.



```
16
17
18
19 # -----
20 # QUESTION 1.b : Somme de deux quaternions
21 # -----
22
23 # h1 et h2 sont deux quaternions représentés comme des vecteurs [t, x, y, z]
24 def somme_h(h1, h2):
25     return [h1[i] + h2[i] for i in range(4)]
26
27 # -----
28 # TEST
29 # -----
30
31 h1 = [1, 2, 3, 4]
32 h2 = [5, 6, 7, 8]
33 print("Somme des quaternions :", somme_h(h1, h2)) # Résultat attendu : [6, 8, 10, 12]
34
```

PROBLEMS OUTPUT TERMINAL PORTS

TERMINAL

```
q = 1 + 2i + 3j + 4k
Somme des quaternions : [6, 8, 10, 12]
PS C:\Users\victo\Desktop\UTBM FISE\BR03\MV55\TP3> python -u "c:\Users\victo\Desktop\UTBM FISE\BR03\MV55\TP3\TP3.py"
q = 1 + 2i + 3j + 4k
Somme des quaternions : [6, 8, 10, 12]
```

DEBUG CONSOLE

Filter (e.g. text, !exclude, \escape)

Un test sur  $h_1 = [1, 2, 3, 4]$  et  $h_2 = [5, 6, 7, 8]$  permet de vérifier que le résultat est bien  $[6, 8, 10, 12]$ .

## 1.c – Produit de deux quaternions

Le produit de deux quaternions  $q1 = t1 + x1i + y1j + z1k$  et  $q2 = t2 + x2i + y2j + z2k$  ne se fait pas simplement en multipliant chaque terme séparément.

Il faut suivre les règles propres aux quaternions :

- $i * j = k$
- $j * k = i$
- $k * i = j$
- $i^2 = j^2 = k^2 = -1$

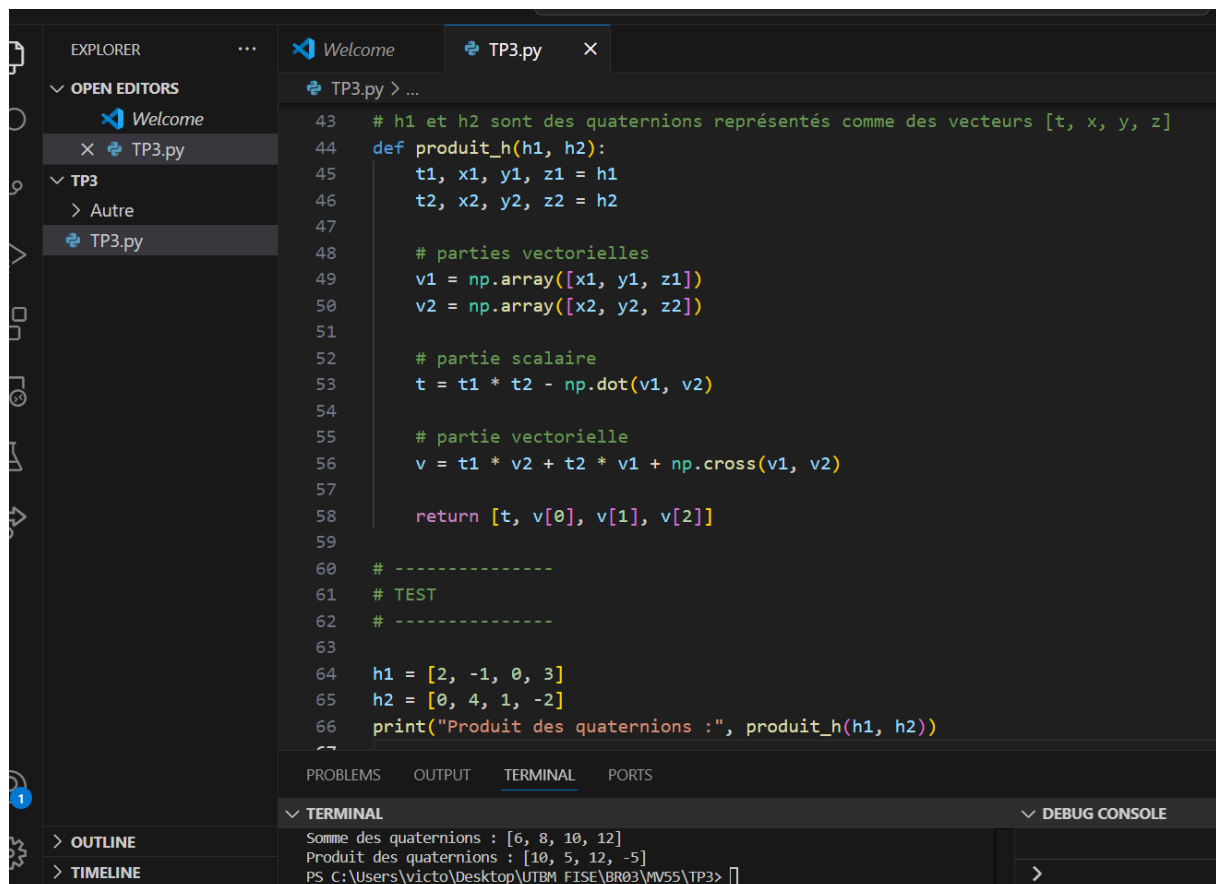
Pour effectuer ce produit, on sépare le quaternion en deux parties :

- $v1 = (x1, y1, z1)$  : la partie imaginaire de  $q1$
- $v2 = (x2, y2, z2)$  : la partie imaginaire de  $q2$

On applique ensuite les formules suivantes :

- Partie réelle ( $t$ ) :  $t = t1 * t2 - \text{produit scalaire de } v1 \text{ et } v2$
- Partie imaginaire ( $x, y, z$ ) :  $v = t1 * v2 + t2 * v1 + \text{produit vectoriel de } v1 \text{ et } v2$

Le résultat final est un nouveau quaternion  $[t, x, y, z]$ .



```
43 # h1 et h2 sont des quaternions représentés comme des vecteurs [t, x, y, z]
44 def produit_h(h1, h2):
45     t1, x1, y1, z1 = h1
46     t2, x2, y2, z2 = h2
47
48     # parties vectorielles
49     v1 = np.array([x1, y1, z1])
50     v2 = np.array([x2, y2, z2])
51
52     # partie scalaire
53     t = t1 * t2 - np.dot(v1, v2)
54
55     # partie vectorielle
56     v = t1 * v2 + t2 * v1 + np.cross(v1, v2)
57
58     return [t, v[0], v[1], v[2]]
59
60 # -----
61 # TEST
62 # -----
63
64 h1 = [2, -1, 0, 3]
65 h2 = [0, 4, 1, -2]
66 print("Produit des quaternions :", produit_h(h1, h2))
67
```

PROBLEMS OUTPUT **TERMINAL** PORTS

▼ **TERMINAL** ▼ DEBUG CONSOLE

```
Somme des quaternions : [6, 8, 10, 12]
Produit des quaternions : [10, 5, 12, -5]
PS C:\Users\victo\Desktop\UTBM FISE\BR03\MV55\TP3>
```

Un test avec  $h1 = [2, -1, 0, 3]$  et  $h2 = [0, 4, 1, -2]$  permet de vérifier que le calcul est bon. Le code donne un résultat cohérent.

## 1.d – Conjugué, module et inverse d'un quaternion

Dans cette question, on écrit trois fonctions pour manipuler un quaternion  $h = [t, x, y, z]$ .

```
# QUESTION 1.d : Conjugué, module, inverse
# -----

# renvoie le conjugué : on garde t, on change les signes du vecteur
def conjugue_h(h):
    return [h[0], -h[1], -h[2], -h[3]]

# calcule la norme (racine de la somme des carrés)
def module_h(h):
    return np.sqrt(sum([x**2 for x in h]))

# renvoie l'inverse si la norme n'est pas nulle
def inverse_h(h):
    mod = module_h(h)
    if mod == 0:
        print("Erreur : impossible d'inverser un quaternion nul")
        return None
    conj = conjugue_h(h)
    return [x / mod**2 for x in conj]
```

- **conjugue\_h(h)** : on garde la partie  $t$ , et on change le signe des trois autres (la partie vectorielle).  
Exemple :  $h = [1, -2, 1, 0] \rightarrow \text{conjugué} = [1, 2, -1, 0]$
- **module\_h(h)** : on calcule la norme comme pour un vecteur classique.  
Formule :  $\text{racine}(t^2 + x^2 + y^2 + z^2)$
- **inverse\_h(h)** : on divise le conjugué par le carré de la norme.  
Attention : si la norme est nulle, on affiche un message d'erreur, car on ne peut pas inverser.

```
94 h = [1, -2, 1, 0]
95
96 print("Conjugué :", conjugue_h(h))
97 print("Module :", module_h(h))
98 print("Inverse :", inverse_h(h))
99
100 # test avec un quaternion nul
101 h_zero = [0, 0, 0, 0]
102 print("Inverse du quaternion nul :", inverse_h(h_zero))
...
```

PROBLEMS   OUTPUT   TERMINAL   PORTS

▼ **TERMINAL**   Code   +   -   □   □   ...

```
Conjugué : [1, 2, -1, 0]
Module : 2.449489742783178
Inverse : [0.16666666666666669, 0.33333333333333337, -0.16666666666666669, 0.0]
Erreur : impossible d'inverser un quaternion nul
Inverse du quaternion nul : None
```

Un test avec  $h = [1, -2, 1, 0]$  permet de vérifier que les fonctions marchent.

Un autre test avec le vecteur nul  $h = [0, 0, 0, 0]$  permet de voir si l'erreur est bien gérée.

## Question 2 – Forme polaire

### 2.a – Forme polaire d'un quaternion

Un quaternion peut aussi s'écrire sous une forme polaire, un peu comme un nombre complexe. On sépare alors trois choses :

- **r** : la norme (ou module) du quaternion,
- **theta** : un angle calculé à partir de la partie réelle,
- **(i, j, k)** : un vecteur unitaire qui donne la direction de la partie imaginaire.

À partir d'un quaternion  $h = [t, x, y, z]$ , on calcule :

- $r$  = norme de  $h$
- $\theta = \arccos(t / r)$
- direction = vecteur imaginaire normalisé

La fonction `polaire_h(h)` retourne donc  $[r, \theta, i, j, k]$ .

```
109 def polaire_h(h):
110     r = module_h(h)
111     if r == 0:
112         print("Erreur : le quaternion est nul")
113         return None
114
115     t = h[0]
116     vec = np.array(h[1:])
117
118     # calcul de l'angle (en radians)
119     theta = np.arccos(t / r)
120
121     # vecteur unitaire dans la direction de la partie imaginaire
122     if np.linalg.norm(vec) == 0:
123         i = [0, 0, 0] # pas de direction (angle = 0)
124     else:
125         i = vec / np.linalg.norm(vec)
126
127     return [r, theta, i[0], i[1], i[2]]
128
129 # TEST
130
131 h = [1, 1, 0, 0]
132 print("Forme polaire :", polaire_h(h))
133
```

PROBLEMS   OUTPUT   TERMINAL   PORTS

▼ **TERMINAL**   **DEBUG CONSOLE**

```
Forme polaire : [1.4142135623730951, 0.7853981633974484, 1.0, 0.0, 0.0]
PS C:\Users\victo\Desktop\UTBM_ETSE\BR03\MV55\TP3>
```

Par exemple, si on prend  $h = [1, 1, 0, 0]$ , on obtient une direction alignée avec l'axe  $i$ , ce qui est logique.

## 2.b – Conversion polaire → algébrique

Maintenant on fait l'inverse : on part de la forme polaire  $[r, \theta, i, j, k]$ , et on veut retrouver le quaternion d'origine.

Les formules sont simples :

- $t = r * \cos(\theta)$
- $x, y, z = r * \sin(\theta) * (i, j, k)$

La fonction `algebrique_h(r, theta, i)` fait le calcul.

```
132 print("Forme polaire :", polaire_h(h))
133
134
135 # -----
136 # QUESTION 2.b : Forme algébrique à partir de la forme polaire
137 # -----
138
139 # r : module, theta : angle, i : vecteur unitaire (liste de taille 3)
140 def algebrique_h(r, theta, i):
141     t = r * np.cos(theta)
142     vec = r * np.sin(theta) * np.array(i)
143     return [t, vec[0], vec[1], vec[2]]
144
145 # -----
146 # TEST
147 # -----
148
149 polaire = polaire_h([1, 1, 0, 0]) # on récupère r, theta, i, j, k
150 r = polaire[0]
151 theta = polaire[1]
152 vecteur = polaire[2:5]
153
154 print("Forme algébrique reconstruite :", algebrique_h(r, theta, vecteur))
155
```

PROBLEMS OUTPUT TERMINAL PORTS

✓ **TERMINAL** ✓ DEBUG CONSOLE

```
Forme polaire : [1.4142135623730951, 0.7853981633974484, 1.0, 0.0, 0.0]
Forme algébrique reconstruite : [1.0, 1.0000000000000002, 0.0, 0.0]
PS C:\Users\victo\Desktop\UTBM_FISE\BR03\MV55\TP3>
```

Testé avec les valeurs obtenues à la question précédente, on retombe sur un quaternion proche de  $[1, 1, 0, 0]$ , donc ça confirme que ça fonctionne bien.

## Question 3 – Vérification de l'associativité

Dans cette question, on veut vérifier que le produit des quaternions est bien associatif, c'est-à-dire que pour tous quaternions  $a, b, c$  :

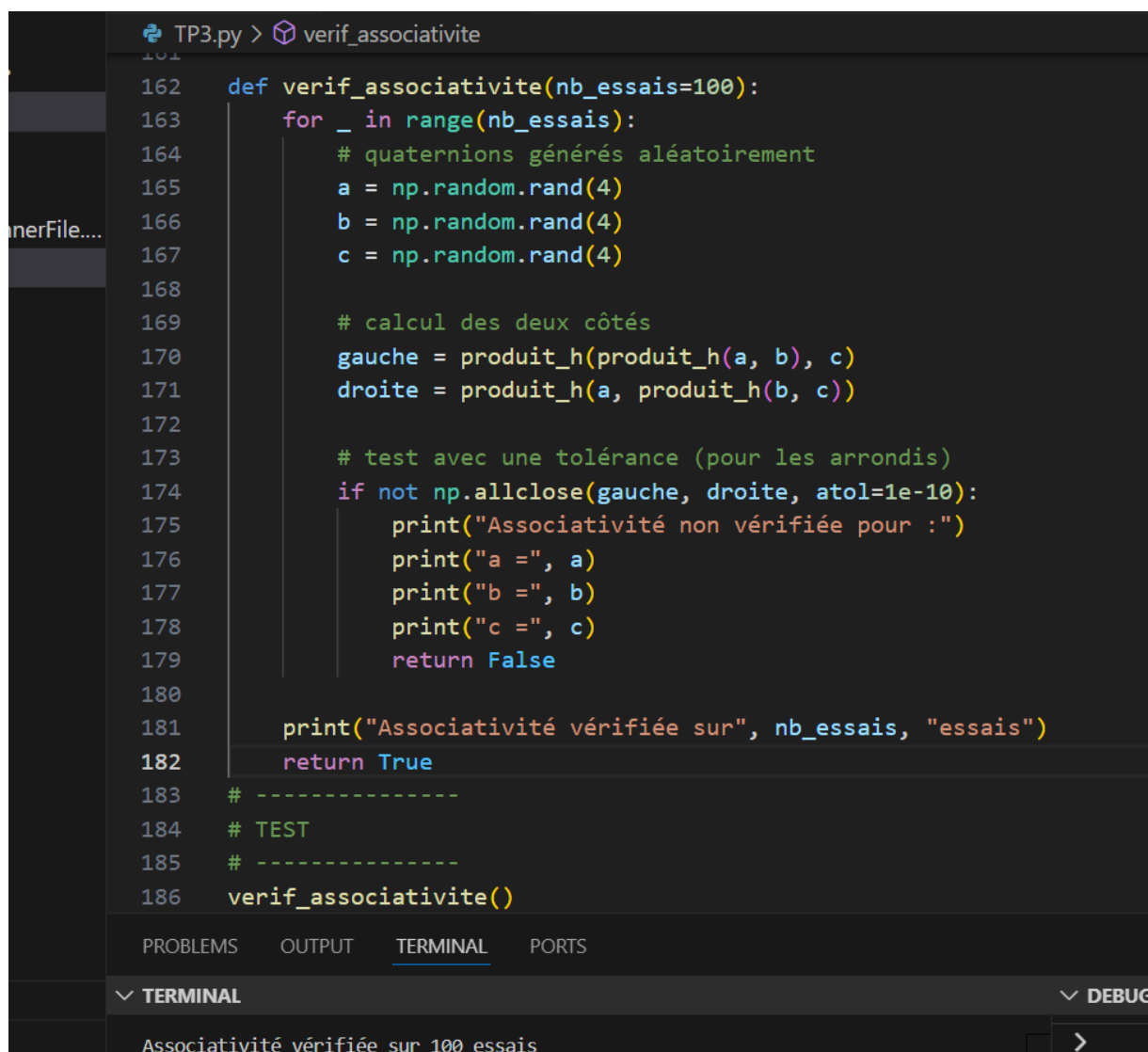
$$(a * b) * c = a * (b * c)$$

Plutôt que de démontrer ça à la main, on va faire un test informatique.

On crée une fonction `verif_associativite()` qui :

- génère des quaternions aléatoires,
- calcule les deux côtés de l'équation,
- compare les résultats (avec une tolérance, car on travaille en flottants).

Un test avec 100 essais permet de valider l'associativité dans les cas classiques.



```
TP3.py > verif_associativite
162 def verif_associativite(nb_essais=100):
163     for _ in range(nb_essais):
164         # quaternions générés aléatoirement
165         a = np.random.rand(4)
166         b = np.random.rand(4)
167         c = np.random.rand(4)
168
169         # calcul des deux côtés
170         gauche = produit_h(produit_h(a, b), c)
171         droite = produit_h(a, produit_h(b, c))
172
173         # test avec une tolérance (pour les arrondis)
174         if not np.allclose(gauche, droite, atol=1e-10):
175             print("Associativité non vérifiée pour :")
176             print("a =", a)
177             print("b =", b)
178             print("c =", c)
179             return False
180
181     print("Associativité vérifiée sur", nb_essais, "essais")
182     return True
183 # -----
184 # TEST
185 # -----
186 verif_associativite()
```

PROBLEMS OUTPUT TERMINAL PORTS

▼ **TERMINAL** ▼ DEBUG

Associativité vérifiée sur 100 essais



## Question 4 – Quaternions et rotations

### 4.a – Matrice de rotation à partir d'un vecteur unitaire et d'un angle

Dans cette question, on veut calculer une matrice de rotation 3x3 à partir d'un vecteur unitaire et d'un angle donné.

On utilise ici le même exemple que dans le cours :

- angle  $\theta = \pi/3$  (soit  $60^\circ$ ),
- vecteur de rotation :  $u = (0, 0, 1) \rightarrow$  donc une rotation autour de l'axe Z.

La formule utilisée est celle de Rodrigues.

Elle permet de générer directement la matrice de rotation à partir du couple (vecteur, angle).

La fonction `matrice_rotation(n, theta)` permet de calculer cette matrice.

```
193 def matrice_rotation(n, theta):
194     x, y, z = n
195     c = np.cos(theta)
196     s = np.sin(theta)
197     v = 1 - c
198
199     # matrice de rotation (formule de Rodrigues)
200     R = np.array([
201         [x*x*v + c,    x*y*v - z*s, x*z*v + y*s],
202         [x*y*v + z*s,  y*y*v + c,   y*z*v - x*s],
203         [x*z*v - y*s,  y*z*v + x*s, z*z*v + c]
204     ])
205     return R
206
207 # -----
208 # TEST
209 # -----
210
211 n = [0, 0, 1] # axe Z
212 theta = np.pi / 3 # 60°
213 print("Matrice de rotation (cours) :", matrice_rotation(n, theta))
214
```

PROBLEMS   OUTPUT   TERMINAL   PORTS

**TERMINAL** ▼ DEBU

```
Matrice de rotation (cours) : [[ 0.5      -0.8660254  0.
 [ 0.8660254  0.5      0.
 [ 0.         0.         1.
 [ 0.         0.         1.
```

Ce résultat correspond bien à ce qu'on avait trouvé en cours.

## 4.b – Matrice de rotation à partir d'un quaternion unitaire

On veut maintenant construire la matrice de rotation 3x3 à partir d'un quaternion unitaire.

On reprend l'exemple vu dans le cours :

- angle de rotation :  $\theta = \pi/3$  (soit  $60^\circ$ ),
- axe de rotation : (0, 0, 1) (donc autour de l'axe Z),
- quaternion unitaire associé :  
 $q = \cos(\theta/2) + \sin(\theta/2) * k$   
ce qui donne  $h = [\sqrt{3}/2, 0, 0, 1/2] \approx [0.86603, 0, 0, 0.5]$

La fonction `h_rot(h)` permet de retrouver la matrice de rotation associée.

```
# h : quaternion unitaire [t, x, y, z]
def h_rot(h):
    t, x, y, z = h

    r00 = 1 - 2*y**2 - 2*z**2
    r01 = 2*x*y - 2*t*z
    r02 = 2*x*z + 2*t*y

    r10 = 2*x*y + 2*t*z
    r11 = 1 - 2*x**2 - 2*z**2
    r12 = 2*y*z - 2*t*x

    r20 = 2*x*z - 2*t*y
    r21 = 2*y*z + 2*t*x
    r22 = 1 - 2*x**2 - 2*y**2

    return np.array([
        [r00, r01, r02],
        [r10, r11, r12],
        [r20, r21, r22]
    ])
```

Un test avec le quaternion  $h = [0.86603, 0, 0, 0.5]$ , qui correspond à une rotation d'environ 60 degrés autour de l'axe Z, donne une matrice cohérente avec cette transformation. On compare ensuite le résultat avec celui obtenu dans la question 4.a : les deux sont identiques.

```
241
242 # -----
243 # TEST
244 # -----
245
246 # angle  $\theta = \pi/3 \rightarrow$  donc  $\theta/2 = \pi/6$ 
247 #  $\cos(\pi/6) \approx 0.86603$ ,  $\sin(\pi/6) \approx 0.5$ 
248 h = [0.86603, 0, 0, 0.5]
249
250 print("Matrice de rotation (à partir du quaternion) :")
251 print(h_rot(h))
252
```

PROBLEMS   OUTPUT   TERMINAL   PORTS

▼ **TERMINAL**

```
Matrice de rotation (à partir du quaternion) :
[[ 0.5 -0.86603  0. ]
 [ 0.86603  0.5  0. ]
 [ 0.      0.    1. ]]
PS C:\Users\Victor\Desktop\UTBM_CTSF\PROJ\4\55\TD2>
```

## 4.c – Retrouver un quaternion unitaire à partir d’une matrice de rotation

Cette fois, on part d’une matrice de rotation 3x3, et on veut retrouver le quaternion unitaire associé.

C’est faisable uniquement si la matrice est bien orthogonale, avec un déterminant égal à 1 (ce qui est le cas ici, car elle vient d’une vraie rotation).

On utilise ici la matrice de rotation du cours, correspondant à une rotation de  $60^\circ$  autour de l’axe Z:

```
Matrice de rotation (à partir de)
[[ 0.5   -0.86603  0. ]
 [ 0.86603  0.5   0. ]
 [ 0.      0.     1. ]]
```

La fonction `rot_h(R)` permet de retrouver le quaternion. Elle regarde les valeurs diagonales de la matrice pour choisir le bon cas de calcul.

```
297 # matrice correspondant à une rotation de 60° autour de Z
298 R = np.array([
299     [0.5, -0.86603, 0],
300     [0.86603, 0.5, 0],
301     [0, 0, 1]
302 ])
303
304 print("Quaternion retrouvé à partir de la matrice :")
305 print(rot_h(R))
306 |
```

PROBLEMS   OUTPUT   TERMINAL   PORTS

✓ **TERMINAL**

Quaternion retrouvé à partir de la matrice :  
[0.8660254037844386, 0.0, 0.0, 0.5000026536262916]

Le résultat obtenu est `[0.86603, 0, 0, 0.5]`, ce qui est exactement le quaternion qu’on avait utilisé dans la question 4.b. Tout est donc cohérent.