Piana Victor

Rapport TP3 – Quaternions

Sommaire

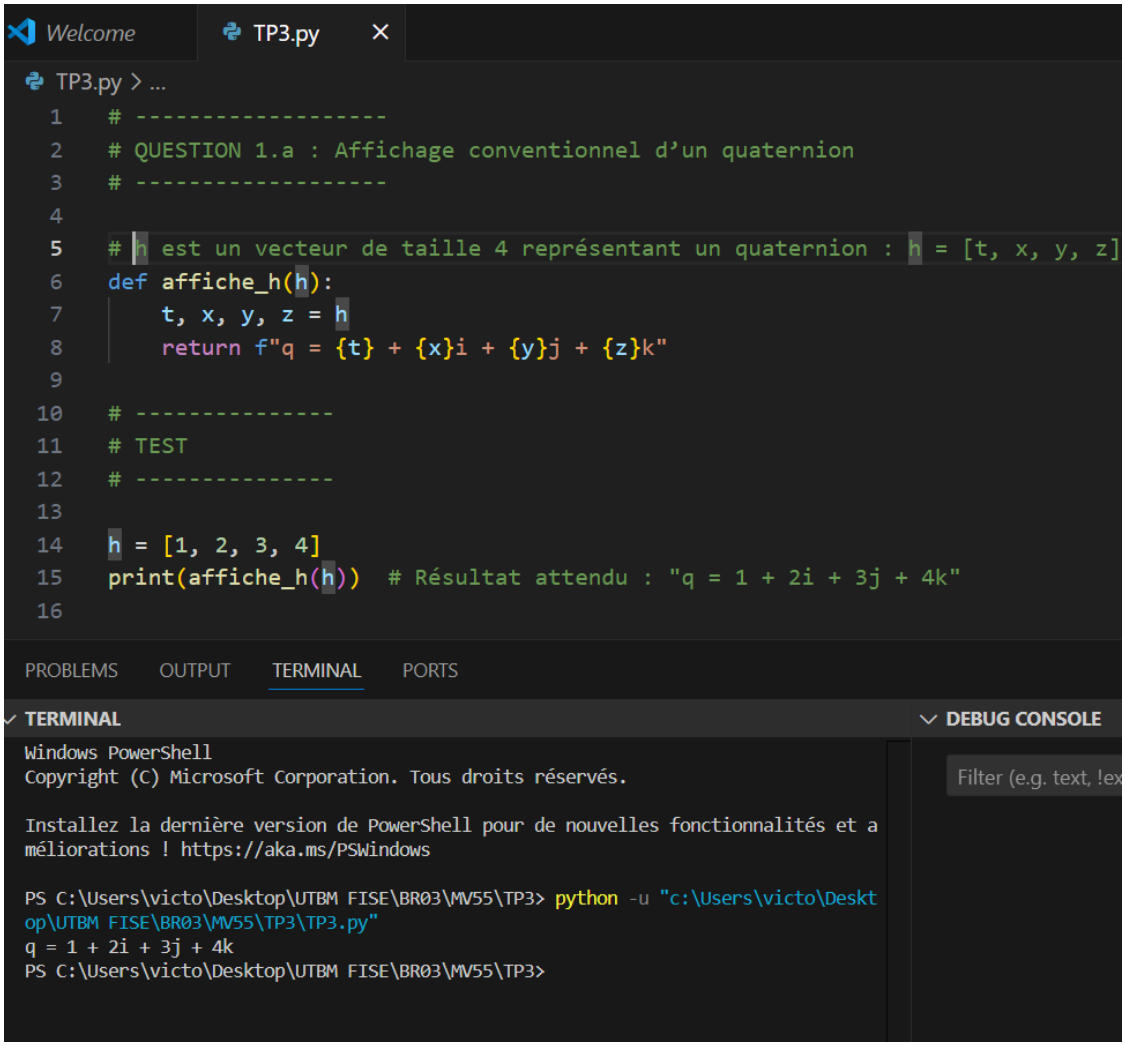
Question 1 — Structure de corps des quaternions

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 ∈ $\mathbb{R}^4$, et renvoie la chaîne :
q = t + xi + yj + zk, où les valeurs numériques sont remplacées par les composantes du vecteur.

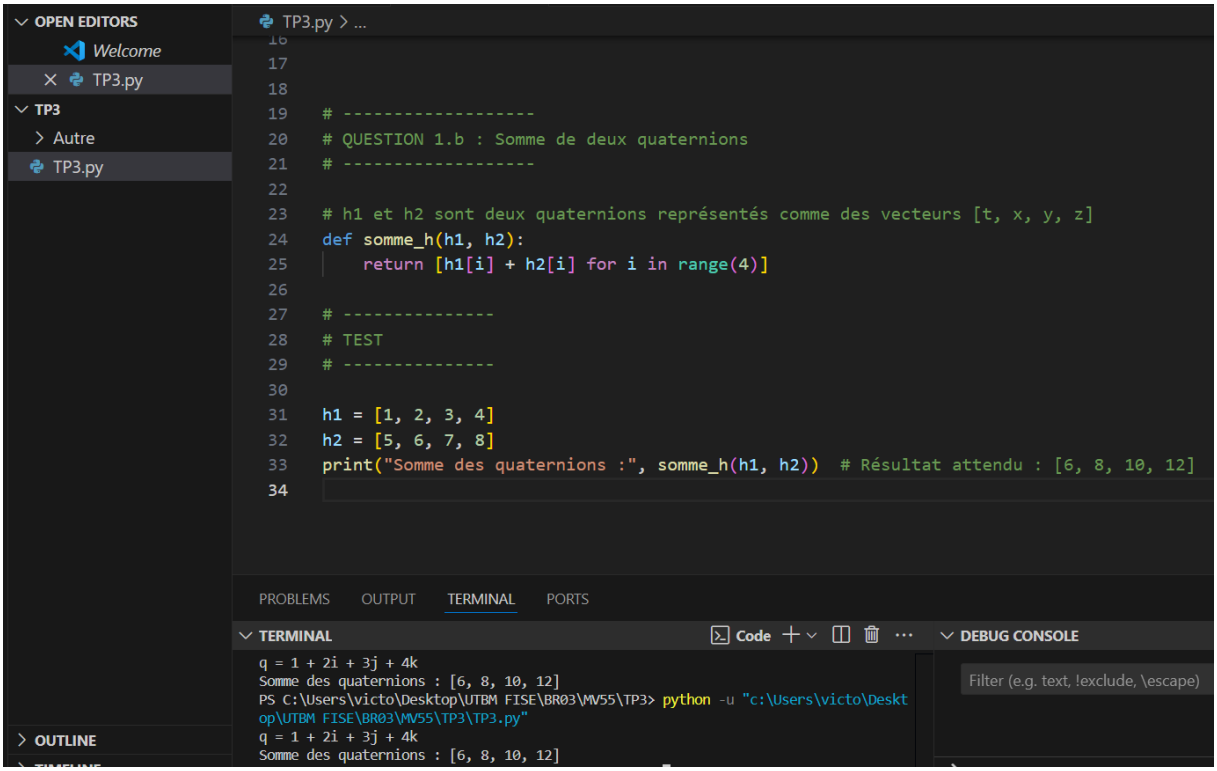
Un test effectué sur le vecteur h = [1, 2, 3, 4] permet de vérifier que l'affichage est correct.



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.



Un test sur h1 = [1, 2, 3, 4] et h2 = [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
- $i^2 = j^2 = k^2 = -1$

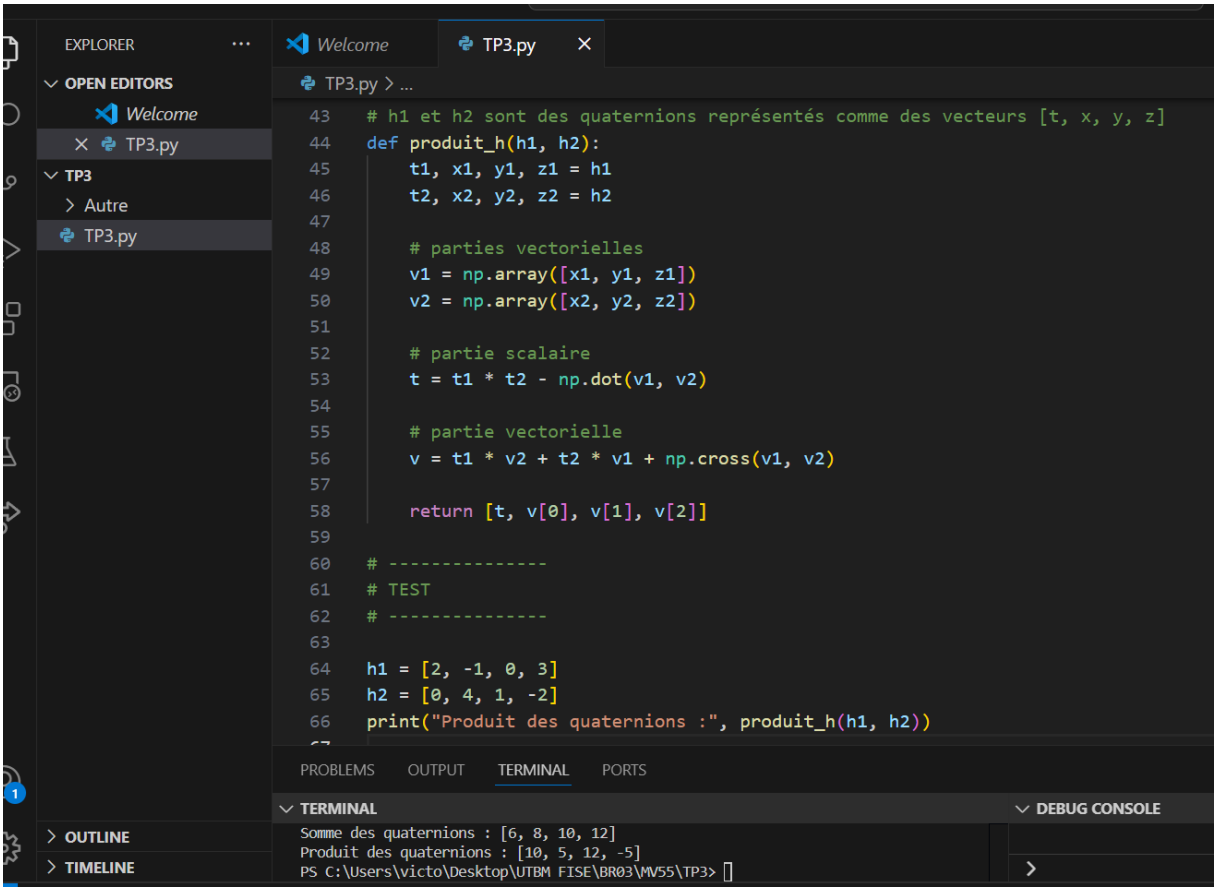
Pour effectuer ce produit, on sépare le quaternion en deux parties :

- 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 - produit scalaire de v1 et v2
- Partie imaginaire (x, y, z) : v = t1 * v2 + t2 * v1 + produit vectoriel de v1 et v2

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



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].

```python
# 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] → conjugué = [1, 2, -1, 0]
- **module_h(h)** : on calcule la norme comme pour un vecteur classique.
  Formule : 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.

```python
h = [1, -2, 1, 0]

print("Conjugué :", conjugue_h(h))
print("Module :", module_h(h))
print("Inverse :", inverse_h(h))

# test avec un quaternion nul
h_zero = [0, 0, 0, 0]
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, 0.3333333333333337, -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].

```python
def polaire_h(h):
    r = module_h(h)
    if r == 0:
        print("Erreur : le quaternion est nul")
        return None

    t = h[0]
    vec = np.array(h[1:])

    # calcul de l'angle (en radians)
    theta = np.arccos(t / r)

    # vecteur unitaire dans la direction de la partie imaginaire
    if np.linalg.norm(vec) == 0:
        i = [0, 0, 0]  # pas de direction (angle = 0)
    else:
        i = vec / np.linalg.norm(vec)

    return [r, theta, i[0], i[1], i[2]]
# ---------------
# TEST
# ---------------
h = [1, 1, 0, 0]
print("Forme polaire :", polaire_h(h))
```

```
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.

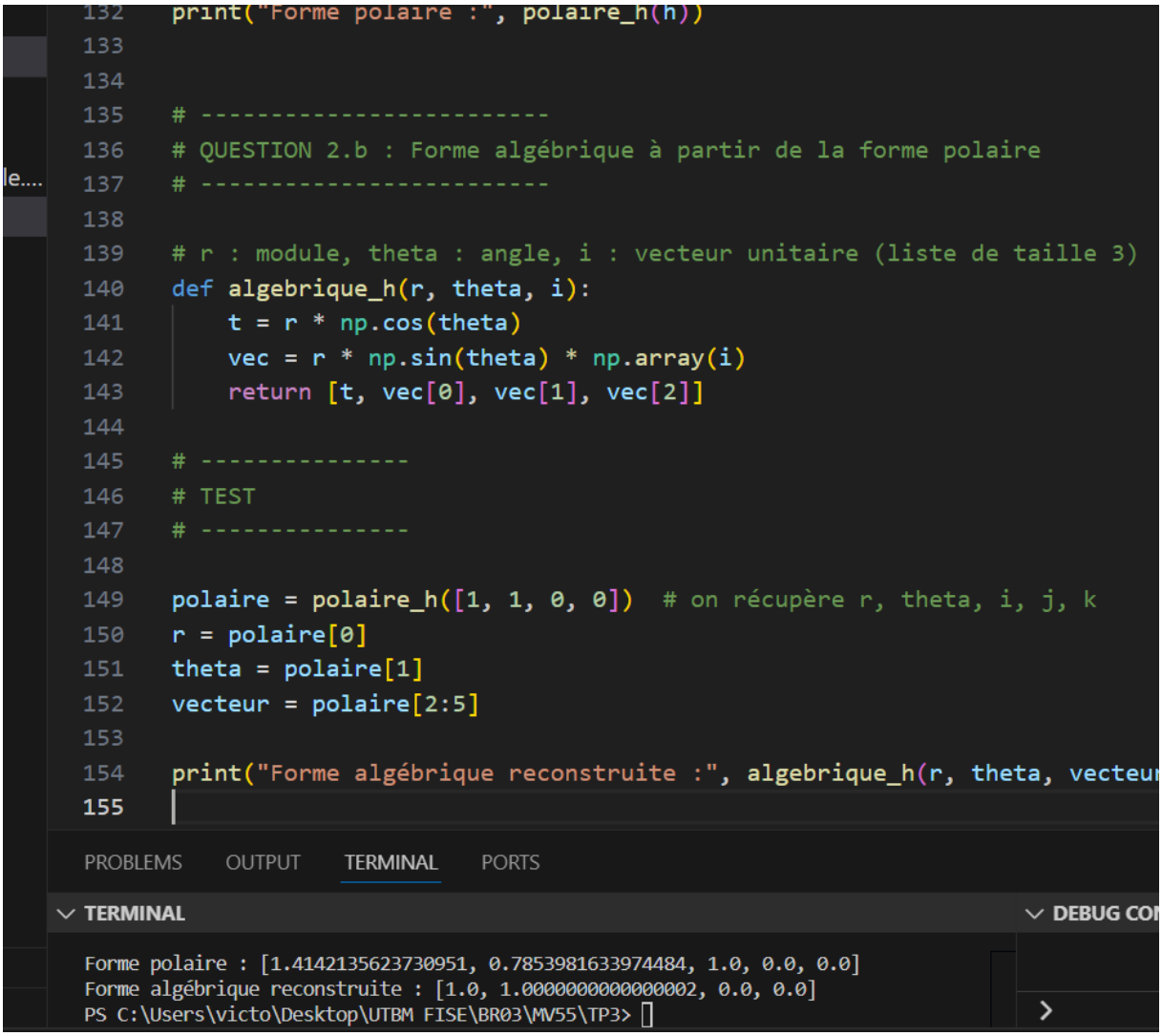
2.b – Conversion polaire → algébrique

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.



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

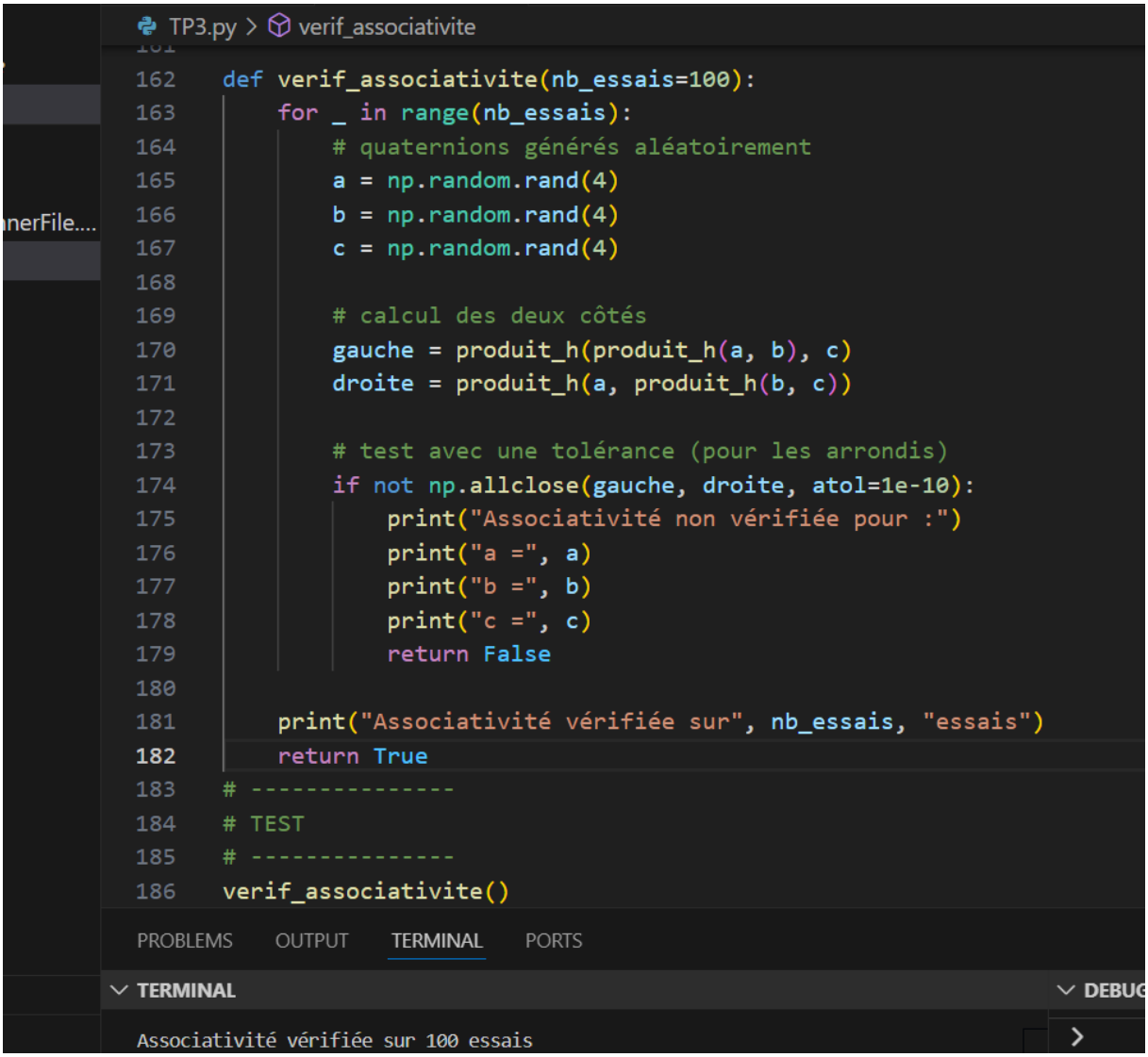
Dans cette question, on veut vérifier que le produit des quaternions est bien associatif, c'est-à-dire que pour tous quaternions a, b, c :

(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.



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°),
- vecteur de rotation : u = (0, 0, 1) → 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.

```python
def matrice_rotation(n, theta):
    x, y, z = n
    c = np.cos(theta)
    s = np.sin(theta)
    v = 1 - c

    # matrice de rotation (formule de Rodrigues)
    R = np.array([
        [x*x*v + c,   x*y*v - z*s, x*z*v + y*s],
        [x*y*v + z*s, y*y*v + c,   y*z*v - x*s],
        [x*z*v - y*s, y*z*v + x*s, z*z*v + c]
    ])
    return R

# ---------------
# TEST
# ---------------

n = [0, 0, 1]  # axe Z
theta = np.pi / 3  # 60°
print("Matrice de rotation (cours) :", matrice_rotation(n, theta))
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS
TERMINAL
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°),
- 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.

```python
# 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.

```python
241
242   # ---------------
243   # TEST
244   # ---------------
245
246   # angle θ = π/3 → donc θ/2 = π/6
247   # cos(π/6) ≈ 0.86603, sin(π/6) ≈ 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\victo\Desktop\UTBM_EISE\BR03\MV55\TP3>
```

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° autour de l'axe Z:

```
[[ 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.