La rotation en 3D grâce aux quaternions

Zakariae El Bouzidi
N° 37323
Option MP

Objectifs :

- Comprendre la représentation quaternionique des rotations en 3D

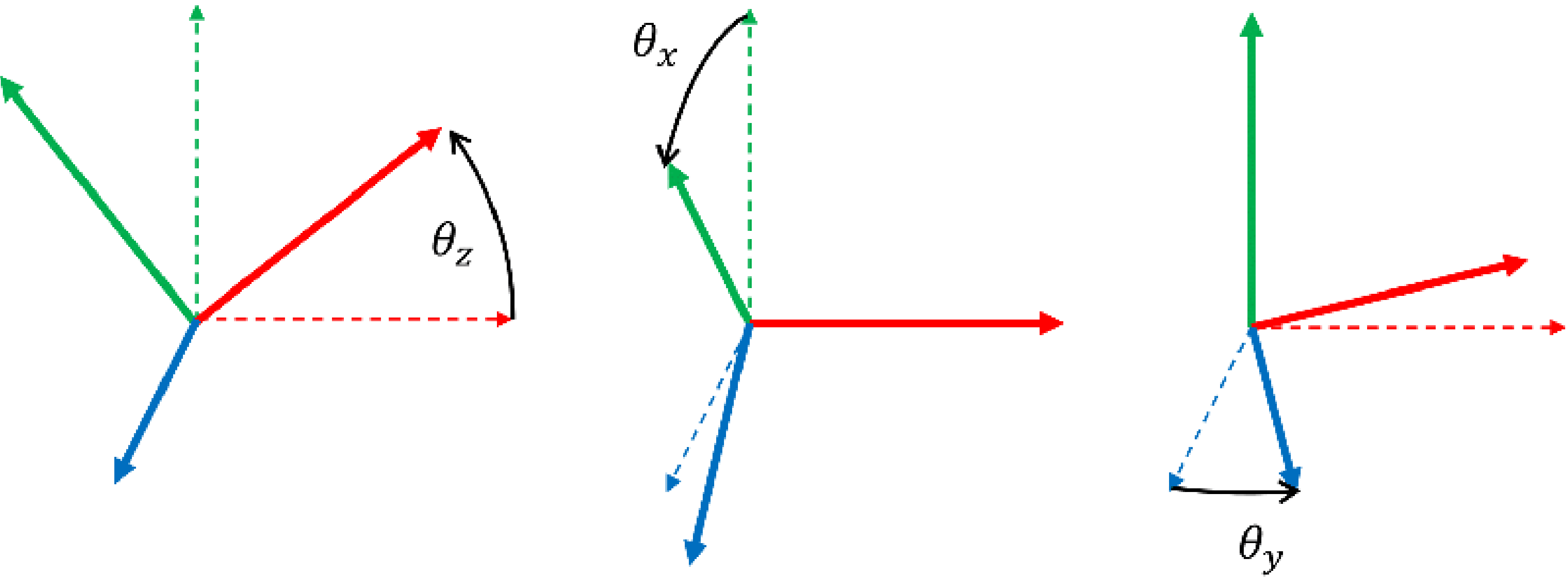
- Analyser les performances de la rotation des quaternions

Sommaire

- **Partie I :** Rotation avec des matrices de rotations.
- **Partie II :** Corps $\mathbb{H}$ des quaternions.
- **Partie III :** Rotation avec des quaternions.
- **Partie IV :** Comparaison des 2 méthodes.

Partie Ⅰ : Rotation avec des matrices de rotations.

- La rotation en 3D est une transformation géométrique qui fait pivoter **un objet** d'un certain **angle *θ*** autour d'**un axe** dans l'espace tridimensionnel.



Partie I : Rotation avec des matrices de rotations.

Dans un espace euclidien à 3 dimensions :

Les matrices de rotations suivantes correspondent à des rotations d'un vecteur , d'un angle θ, autour des axes x, y et z (respectivement) :

$$R_x(\theta) = \begin{matrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{matrix}$$

$$R_y(\theta) = \begin{matrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{matrix}$$

$$R_z(\theta) = \begin{matrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{matrix}$$

On peut aussi construire **une matrice de rotation suivant un vecteur directeur de l'axe de rotation:**

$$P * R_x(\theta) * P^T$$

Ainsi on déduit la rotation d'un vecteur v :

$$P * R_x(\theta) * P^T * \mathrm{v}$$

On compte alors 90 multiplications et 60 additions.

Former la matrice de passage

```python
def base(x, y, z):
    matrice = np.zeros((3, 3))
    matrice[:, 0] = [x, y, z]
    if z != 0:
        matrice[:, 1] = [1, 0, 0]
        matrice[:, 2] = [0, 1, 0]
    elif y != 0:
        matrice[:, 1] = [1, 0, 0]
        matrice[:, 2] = [0, 0, 1]
    else:
        matrice[:, 1] = [0, 1, 0]
        matrice[:, 2] = [0, 0, 1]

    #on vérifie le déterminant
    determinant = np.linalg.det(matrice)
    if determinant < 0:
        #si le déterminant est négatif, on essaie de transposer les colonnes
        matrice[:, [1, 2]] = matrice[:, [2, 1]]
        determinant = np.linalg.det(matrice)
        if determinant < 0:
            matrice[:, 1] = -matrice[:, 1]
    return matrice
```

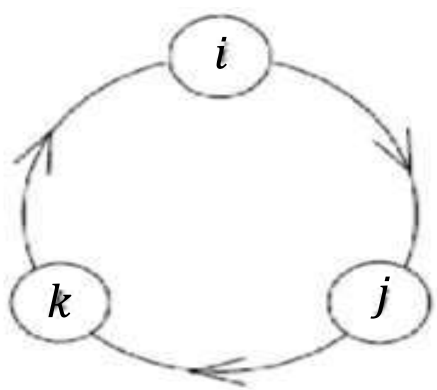
```python
def gram_schmidt(a1, a2, a3):
    #première colonne
    f1 = a1
    e1 = f1 / np.linalg.norm(f1)
    #deuxième colonne
    f2 = a2 - np.vdot(e1, a2) * e1
    e2 = f2 / np.linalg.norm(f2)
    #troisième colonne
    f3 = a3 - np.vdot(e1, a3) * e1 - np.vdot(e2, a3) * e2
    e3 = f3 / np.linalg.norm(f3)

    return np.column_stack((e1, e2, e3))
```

Partie II : Corps $\mathbb{H}$ des quaternions.

Les quaternions sont présentés par l'ensemble :

$$\mathbb{H} = \{a + bi + cj + dk : a, b, c, d \in R\}$$

$$o\text{ù } i^2 = j^2 = k^2 = -1$$



Structure de corps :

- $\forall q_1, q_2 \in \mathbb{H}$, $q_1 + q_2$ et $q_1 \cdot q_2$ sont également dans $\mathbb{H}$.
- $0 + 0i + 0j + 0k$, qui est $0_{\mathbb{H}}$
- $1 + 0i + 0j + 0k$, qui est $1_{\mathbb{H}}$
- Pour $q = a + bi + cj + dk$, $q' = a - bi - cj - dk$
- Pour $q = a + bi + cj + dk$, $q^{-1} = \frac{a - bi - cj - dk}{a^2 + b^2 + c^2 + d^2}$

- Nous pouvons penser à un quaternion comme ayant une partie scalaire (nombre) et une partie vectorielle:
$$v_0 + v_1\boldsymbol{i} + v_2\boldsymbol{j} + v_3\boldsymbol{k} = \ v_0, \boldsymbol{v}$$

- On définit alors l'isomorphisme :

$$\phi : \ \mathbb{H} \rightarrow \mathbb{R} \times \mathbb{R}^3$$

$$\phi(a + bi + cj + dk) = \ (a, v)$$

Où : $a \ \in \mathbb{R}\ et\ v = (b, c, d) \ \in \mathbb{R}^3$

- Pour définir le produit des quaternions d'une autre manière :

$$(v_0\,,\boldsymbol{v}\,)\otimes(w_0,\boldsymbol{w}) = (v_0w_0 - \boldsymbol{v}\,.\,\boldsymbol{w}, v_0\boldsymbol{w} + w_0\boldsymbol{v} \;+\; \boldsymbol{v}\,\times\boldsymbol{w})$$

$$\text{Produit scalaire} = 3 \text{ multiplications et } 2 \text{ additions}$$
$$\text{Produit vectoriel} = 6 \text{ multiplications et } 3 \text{ additions}$$

On compte alors 16 multiplications et 8 additions

Partie Ⅲ : Rotation avec des quaternions.

On pose $\overrightarrow{u} = (u_x, u_y, u_z)$ un vecteur unitaire.

$$q = [\cos\left(\frac{\theta}{2}\right), (u_x, u_y, u_z)\sin\left(\frac{\theta}{2}\right)]$$

$$\vec{p} = (p_x, p_y, p_z)$$

On peut alors en déduire :

$$q \otimes [0, \vec{p}] \otimes q^{-1} = \left[0, \overrightarrow{p'}\right]$$

qui utilise ainsi 48 multiplications et 24 additions.

Partie Ⅲ : Rotation avec des quaternions.

```
import numpy as np
def NormAxe(X):
    norm = np.vdot(X, X)
    return X / np.sqrt(norm)

def ConjugueQuaternion(q):
    w, x, y, z = q
    return np.array([w, -x, -y, -z])

def RotationDuVecteur(vecteur,axe,angle):
    (a,b,c) = (vecteur[0],vecteur[1],vecteur[2])
    (x,y,z) = (axe[0],axe[1],axe[2])
    quaternion = np.array([x,y,z])
    angle_radian = np.radians(angle)
    axe = NormAxe(quaternion)
    v_0 = np.cos(angle_radian / 2)
    qx, qy, qz = axe * np.sin(angle_radian / 2)
    qconj = np.array([-qx,-qy,-qz])
    v = np.array([qx,qy,qz])
    w = np.array([a,b,c])
    A = (v_0*0 - np.vdot(v,w),[v_0*w + 0*v + np.cross(v,w)])
    B = (A[0]*v_0 - np.vdot(A[1],qconj),[A[0]*np.array(qconj) + v_0*np.array(A[1]) + np.cross(A[1],qconj)])
    return B[1:][0][0].T

vecteur = np.array([2,1,1])
axe_rotation = np.array([1/np.sqrt(3),1/np.sqrt(3),1/np.sqrt(3)])
angle = 180
```

```
Résultat après rotation
[[0.66666667]
 [1.66666667]
 [1.66666667]]
```

Partie IV : Comparaison des 2 méthodes.

La différence de temps entre les deux

```python
nombre_repetitions = 100
temps_execution_rotation_matricielle = []
temps_execution_rotation_quaternion = []
for _ in range(nombre_repetitions):
    vecteur = 2 * np.random.rand(3) - 1
    axe_rotation = 2 * np.random.rand(3) - 1
    angle = (2 * np.random.rand() - 1) * 2 * np.pi

    #Matrice de rotation
    debut_temps_1 = time.time()
    for i in range(nombre_repetitions):
        resultat1 = RotationMatricielle(vecteur, axe_rotation, angle)
    fin_temps_1 = time.time()
    temps_execution_rotation_matricielle.append((fin_temps_1 - debut_temps_1) / nombre_repetitions)

    #Rotation par quaternion
    debut_temps_2 = time.time()
    for i in range(nombre_repetitions):
        resultat2 = RotationDuVecteur(vecteur, axe_rotation, angle)
    fin_temps_2 = time.time()
    temps_execution_rotation_quaternion.append((fin_temps_2 - debut_temps_2) / nombre_repetitions)

temps_moyen_rotation_matricielle = np.mean(temps_execution_rotation_matricielle)
temps_moyen_rotation_quaternion = np.mean(temps_execution_rotation_quaternion)
```

Partie IV : Comparaison des 2 méthodes.

Résultats:

```
Temps moyen avec les matrices de rotation
0.0000849284 secondes
Temps moyen pour la rotation avec les
quaternions
0.0000391272 secondes
```

ANNEXES: Script des rotations matricielles

```python
import numpy as np
import time
# SCRIPT ROTATION MATRICIELLE
def base(x, y, z):
    matrice = np.zeros((3, 3))
    matrice[:, 0] = [x, y, z]
    if z != 0:
        matrice[:, 1] = [1, 0, 0]
        matrice[:, 2] = [0, 1, 0]
    elif y != 0:
        matrice[:, 1] = [1, 0, 0]
        matrice[:, 2] = [0, 0, 1]
    else:
        matrice[:, 1] = [0, 1, 0]
        matrice[:, 2] = [0, 0, 1]

    determinant = np.linalg.det(matrice)
    if determinant < 0:
        matrice[:, [1, 2]] = matrice[:, [2, 1]]
        determinant = np.linalg.det(matrice)
        if determinant < 0:
            matrice[:, 1] = -matrice[:, 1]
    return matrice

def gram_schmidt(a1, a2, a3):
    f1 = a1
    e1 = f1 / np.linalg.norm(f1)
    f2 = a2 - np.vdot(e1, a2) * e1
    e2 = f2 / np.linalg.norm(f2)
    f3 = a3 - np.vdot(e1, a3) * e1 - np.vdot(e2, a3) * e2
    e3 = f3 / np.linalg.norm(f3)
    return np.column_stack((e1, e2, e3))
```

Script des rotations matricielles (suite) :

```python
def gram_schmidt(a1, a2, a3):
    f1 = a1
    e1 = f1 / np.linalg.norm(f1)
    f2 = a2 - np.vdot(e1, a2) * e1
    e2 = f2 / np.linalg.norm(f2)
    f3 = a3 - np.vdot(e1, a3) * e1 - np.vdot(e2, a3) * e2
    e3 = f3 / np.linalg.norm(f3)
    return np.column_stack((e1, e2, e3))

def RotationMatricielle(vecteur, axe_rotation, angle):
    axe_rotation = axe_rotation / np.linalg.norm(axe_rotation)
    P1 = base(axe_rotation[0], axe_rotation[1], axe_rotation[2])
    P = gram_schmidt(P1.T[0], P1.T[1], P1.T[2])
    r_B = np.array([[1, 0, 0],
                    [0, np.cos(angle), -np.sin(angle)],
                    [0, np.sin(angle), np.cos(angle)]])
    A = np.dot(P, np.dot(r_B, P.T))
    return np.dot(A, vecteur).reshape(-1, 1)

```

Script des quaternions:

```python
import numpy as np

def NormAxe(X):
    # Fonction de normalisation de l'axe de rotation
    # Pour ne pas se restreindre à x^2+y^2+z^2=1
    norm = np.sum(X**2)

    return X /np.sqrt(norm)

def MultiplierQuaternion(q1, q2):

    w1, x1, y1, z1 = q1
    w2, x2, y2, z2 = q2
    w = w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2
    x = w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2
    y = w1 * y2 - x1 * z2 + y1 * w2 + z1 * x2
    z = w1 * z2 + x1 * y2 - y1 * x2 + z1 * w2

    return np.array([w, x, y, z])

def ConjugueQuaternion(q):
    #w dans notre cas représente la partie réelle (np.cos(angle/2))
    w, x, y, z = q
    return np.array([w, -x, -y, -z])
```

Script de quaternions : (suite)

```
#Arguments :
    #Vecteur sur lequel on applique la rotation : (a,b,c)
    #Vecteur directeur de l'axe de rotation : (x,y,z)
    #Angle de rotation : angle
def RotationDuVecteur(vecteur,axe,angle):


    #normalisation de l'axe de rotation
    axe = NormAxe(axe)


    # Création du quaternion de rotation
    qw = np.cos(angle/2)

    qx, qy, qz = axe * np.sin(angle/2)

    quaternion = np.array([qw, qx, qy, qz])

    # Conversion du vecteur en quaternion
    VectQuaternion = np.concatenate(([0], vecteur))

    # Rotation du vecteur avec le quaternion (q*p*q-1)
    VectQuaternionApresRotation=MultiplierQuaternion(quaternion,MultiplierQuaternion(VectQuaternion,ConjuqueQuaternion(quaternion)))
```

Script de calcul du temps:

```python
import numpy as np
import time
from ROTATIONMATRICEVECTEURR import RotationMatricielle
from RotationQuatPoint import RotationDuVecteur
np.random.seed(3)


nombre_repetitions = 100
temps_execution_rotation_matricielle = []
temps_execution_rotation_quaternion = []
for _ in range(nombre_repetitions):
    vecteur = 2 * np.random.rand(3) - 1
    axe_rotation = 2 * np.random.rand(3) - 1
    angle = (2 * np.random.rand() - 1) * 2 * np.pi

    #Matrice de rotation
    debut_temps_1 = time.time()
    for i in range(nombre_repetitions):
        resultat1 = RotationMatricielle(vecteur, axe_rotation, angle)
    fin_temps_1 = time.time()
    temps_execution_rotation_matricielle.append((fin_temps_1 - debut_temps_1) / nombre_repetitions)

    #Rotation par quaternion
    debut_temps_2 = time.time()
    for i in range(nombre_repetitions):
        resultat2 = RotationDuVecteur(vecteur, axe_rotation, angle)
    fin_temps_2 = time.time()
    temps_execution_rotation_quaternion.append((fin_temps_2 - debut_temps_2) / nombre_repetitions)

temps_moyen_rotation_matricielle = np.mean(temps_execution_rotation_matricielle)
temps_moyen_rotation_quaternion = np.mean(temps_execution_rotation_quaternion)

print("Temps moyen avec les matrices de rotation ")
print("{:.5f} secondes".format(temps_moyen_rotation_matricielle))
print("Temps moyen pour la rotation avec les quaternions")
print("{:.5f} secondes".format(temps_moyen_rotation_quaternion))
```