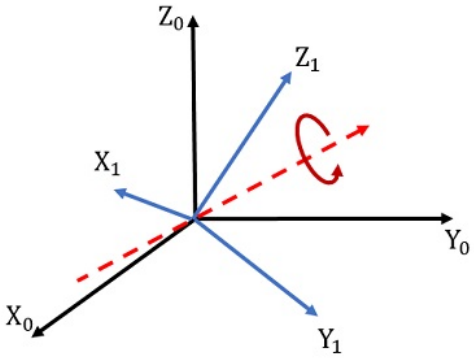
Spatial Rotation with Quaternions

Dimitriy Georgiev

May 11, 2021

1 Assignment Description

The assignment requires the implementation of a programme capable of rotating a point $p = (p_x, p_y, p_z)$ by an angle α around a vector $\vec{u} = (u_x, u_y, u_z)$ in $\mathbb{R}^3$. The document presents three different ways of computing the rotated vector from quaternion representation and compares the execution time of each method. Surprisingly, the performance of the three methods implemented in Python was contradictory to the information provided in Wikipedia. The method that was called "less efficient" had the shortest execution time. Perhaps, comparing the methods at the scale of a single vector rotation is unreasonable and insufficient, but still interesting to see.



2 Background

According to Euler's rotation theorem [1], any rotation or sequence of rotations of an object about a fixed point, in 3-dimensional space, is equivalent to a single rotation by a given angle θ about an axis (*Euler axis*, typically

represented by a unit vector $\vec{u}$) that runs through the fixed point.
There are several ways one can represent rotation, including axis-angle, rotation matrices, and quaternions. This document focuses specifically on quaternions and presents an implementation in Python.

Any vector ($\in \mathbb{R}^3$) can be expressed by a quaternion whose real part is equal to zero. The coefficients in front of the imaginary numbers i, j, k represent the three Cartesian coordinates (x, y, z). Based on an extension of *Euler's formula*, a rotation of angle θ about an axis $\vec{u}$, can be expressed as:

$$q = q_{real} + q_x i + q_y j + q_z k = \cos\frac{\theta}{2} + (u_x i + u_y j + u_z k)\sin\frac{\theta}{2}$$

For numerical computation, quaternions can be stored in an array with 4 elements, compared to 9 when using rotation matrices. Additionally, when combining several successive rotations, quaternions are more efficient to normalize and save 17 floating point operations compared to rotation matrices. Tracking the orientation of a rigid body, for example, is about chaining rotations, and quaternions would offer a reduction in memory and number of arithmetic operations needed. On the other hand, rotation matrices are more efficient to compute the rotation of a vector which requires only 9 multiplications and 6 additions - 15 operations in total. The cost of computing rotated vector from a quaternion representation requires 15 multiplications and 15 additions - 30 operations in total, based on *Rodrigues' formula* given by $\vec{v}_{new} = \vec{v} + 2\vec{r} \times (\vec{r} \times \vec{v} + w\vec{v})$. The quaternion can be converted to a rotation matrix as well, but that would require additional 9 multiplications and 6 additions. [1]

The three methods that were tested are the following:

- Rotation Matrix:

  $$\vec{v}_{new} = R \cdot \vec{v}$$

  where the rotation matrix is given by:

  $$R = 2 \cdot \begin{bmatrix} q_{real}^2 + q_x^2 - 0.5 & q_x q_y - q_{real} q_z & q_{real} q_y + q_x q_z \\ q_{real} q_z + q_x q_y & q_{real}^2 + q_y^2 - 0.5 & q_y q_z - q_{real} q_x \\ q_x q_z - q_{real} q_y & q_{real} q_x + q_y q_z & q_{real}^2 + q_z^2 - 0.5 \end{bmatrix}$$

- Rodrigues' Formula:

  $$\vec{v}_{new} = \vec{v} + 2\vec{r} \times (\vec{r} \times \vec{v} + w\vec{v})$$

  where w is the scalar real part (q_{real}) and $\vec{r}$ is the imaginary part (q_x, q_y, q_z) of the quaternion.

- Conjugation of $\vec{v}$ by q:

  $$\vec{v}_{\text{new}} = q\vec{v}q^*$$

3 Implementation in Python

```
import numpy as np
from numpy.linalg import norm
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def getUnitVector(vector):
    '''Convert a vector in R3 to a unit vector'''
    vector_hat = vector / norm(vector)
    return vector_hat

def quaternionFromEulerAxis(theta, vector):
    '''Create a quaternion representation of the
       desired rotation'''
    q_real = np.cos(theta/2)
    q_x = vector[0] * np.sin(theta/2)
    q_y = vector[1] * np.sin(theta/2)
    q_z = vector[2] * np.sin(theta/2)
    return np.array([q_real, q_x, q_y, q_z])

def conjugateQuaternion(q):
    '''Calculate a quaternion conjugate'''
    w, x, y, z = q
    return (w, -x, -y, -z)

def rotationMatrixFromQuaternion(q):
    '''Calculate rotation matrix from a quaternion'''
    R = 2 * np.array([
        [q[0]** 2 + q[1]** 2 - 0.5, q[1] * q[2] - q[0]
           * q[3], q[0] * q[2] + q[1] * q[3]],
        [q[0] * q[3] + q[1] * q[2], q[0]** 2 + q[2]** 2
            - 0.5, q[2] * q[3] - q[0] * q[1]],
        [q[1] * q[3] - q[0] * q[2], q[0] * q[1] + q[2]
           * q[3], q[0]** 2 + q[3]** 2 - 0.5]
    ])
    return R

def quaternionMultiplication(q1, q2):
    '''Multiplication of two quaternions'''
```

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

def crossProduct(v1, v2):
    '''Cross product of two 3D vectors'''
    a1, a2, a3 = v1
    b1, b2, b3 = v2
    return (a2*b3 - a3*b2, a3*b1 - a1*b3, a1*b2 - a2*b1
        )

p = np.array([1, 0, 0])
U = np.array([0, 1, 0])
theta = 90
theta = np.radians(theta)

u = getUnitVector(U)
q = quaternionFromEulerAxis(theta, u)
```

...

- %%timeit
  $R = rotationMatrixFromQuaternion(q)$
  $p_{new} = R.dot(p)$

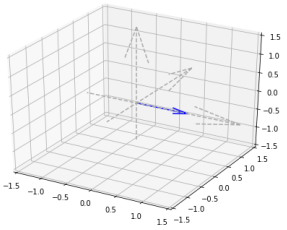
  Execution time: 17.5 µs ± 425 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

- %%timeit
  $A = 2 * q[1:4]$
  $B = crossProduct(q[1:4], p) + q[0] * p$
  $p_{new} = p + crossp(A, B)$

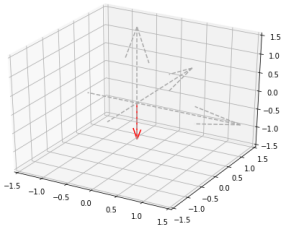
  Execution time: 25.1 µs ± 568 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)

- %%timeit
  $p_{new} = quaternionMultiplication(quaternionMultiplication(q, p), conjugateQuaternion(q))$

Execution time: 13.8 µs ± 109 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)



(a) Before rotation



(b) After 90°

Point	Rotation Axis	Angle	Output
(1,0,0)	(0,1,0)	120	(-0.5,0,-0.866)
(1,1,0)	(0,1,0)	120	(-0.5,1,-0.866)
(1,1,1)	(0,1,0)	90	(1,1,-1)
(2,0,0)	(0,1,0)	90	(0,0,-2)
(2,1,0)	(0,1,0)	45	(1.414,1,-1.414)
(2,1,1)	(0,1,0)	45	(2.121,1,-0.707)
(3,0,0)	(0,1,0)	15	(2.898,0,-0.777)
(3,1,0)	(0,1,0)	15	(2.898,1,-0.777)

References

[1] Quaternions and spatial rotation - Wikipedia *en.wikipedia.org*
`https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation`