# Rigid-Body Orientation Using Unit Quaternions

Emilio Justin - 13524043
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13524043@stei.itb.ac.id , emilio.justin54@gmail.com

***Abstract*—Rigid-body orientation in three-dimensional space is essential for robotics, computer graphics, and aerospace applications. Traditional Euler angles suffer from gimbal lock, while rotation matrices are computationally redundant and numerically unstable. This paper presents a comprehensive mathematical study of unit quaternions as an alternative representation. The theoretical foundations of quaternion algebra are developed, establishing the correspondence with SO(3) through axis-angle representation and the sandwich product. To validate these theoretical properties, computational experiments are conducted using Python implementations for unit quaternions representation, with rotation matrices included as a reference representation to support empirical validation, examining numerical stability and computational efficiency.**

***Keywords*—Quaternion, Rigid-Body, Rotation Matrix, Rotation, Special Orthogonal**

## I. Introduction

The orientation of objects in three-dimensional space is a fundamental topic in linear algebra and geometry, with important applications in computer graphics, robotics, aerospace engineering, and computer vision. Accurately modeling the orientation of a rigid body is essential for describing rotational behavior while preserving geometric properties such as distances and angles. A mathematically consistent framework is therefore required to ensure that rotations remain rigid and well-defined in three-dimensional space.

Traditional approaches to modeling orientation often rely on Euler angles, which describe rotation as a sequence of axis-aligned rotations. Although intuitive, Euler angles suffer from inherent limitations, most notably gimbal lock, a singularity that results in the loss of one rotational degree of freedom. This limitation can cause ambiguity and instability when describing rigid-body orientation, particularly in scenarios involving continuous or compounded rotations.

Unit quaternions offer a mathematically robust approach for describing three-dimensional orientation. By extending complex numbers into four dimensions and enforcing a unit-norm constraint, unit quaternions represent rotations as normalized algebraic entities. This formulation avoids singularities, preserves orthogonality, and allows rotations to be composed efficiently through quaternion multiplication. Consequently, unit quaternions have become a widely adopted tool for modeling rigid-body orientation in both theoretical and computational contexts.

This paper focuses on rigid-body orientation using unit quaternions from linear algebra and geometric perspective. The discussion is restricted to rotational motion in three-dimensional Euclidean space, without considering translational motion or dynamic control. By examining the mathematical foundations of unit quaternions and their role in describing rigid-body orientation, this paper aims to provide a clear and accessible understanding of their effectiveness in modeling three-dimensional rotations.

## II. Theoretical Framework

### A. Rigid-Body Rotational Motion in $\mathbb{R}^3$

A rigid motion of an object is a motion which preserves distance between points [1]. Consequently, any admissible motion of a rigid body must preserve both distances and angles between points. Transformations with this property are known as rigid-body transformations. When translational motion is excluded, such transformations reduce to pure rotations.

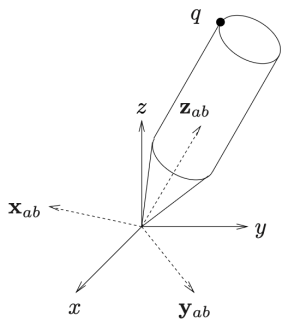


Fig. 1: Rotation of a rigid object about a point. The dotted coordinate frame is attached to the rotating rigid body.
*source: Adapted from [1]*

Rigid-body orientation describes the rotational configuration of an object relative to a reference coordinate frame. Formally, orientation can be defined as the relative alignment between a body-fixed coordinate frame and an inertial frame. This alignment determines how the rigid body is rotated in space, independently of its position. Let $A$ be the inertial frame, $B$ the body frame, and $\mathbf{x}_{ab}, \mathbf{y}_{ab}, \mathbf{z}_{ab} \in \mathbb{R}^3$ the coordinates of the principal axes of $B$ relative to $A$ (see Fig. 1). Stacking these coordinate vectors next to each other, we define a $3 \times 3$ matrix:

$$R_{ab} = \begin{bmatrix} \mathbf{x}_{ab} & \mathbf{y}_{ab} & \mathbf{z}_{ab} \end{bmatrix}$$

We call a matrix constructed in this manner a *rotation matrix*: every rotation of the object relative to the ground corresponds to a matrix of this form [1].

A rotation matrix has two key properties that follow from its construction. Let $R \in \mathbb{R}^{3\times3}$ be a rotation matrix and $r_1, r_2, r_3 \in \mathbb{R}^3$ be its columns. Since the columns of $R$ are mutually orthonormal, the two properties are

$$RR^T = R^T R = I$$

and

$$\det(R) = +1$$

The set of all $3 \times 3$ matrices which satisfy these two properties is denoted $SO(3)$. The notation $SO$ abbreviates *special orthogonal*. Special refers to the fact that $\det(R) = +1$ rather than $\pm1$ [1]. We define the space of rotation matrices in $\mathbb{R}^{3\times3}$ by

$$SO(3) = \{R \in \mathbb{R}^{3\times3} : RR^T = R^T R = I, \det(R) = +1\}$$

Rotation matrices also define linear transformations between coordinate frames [1]. Let $\mathbf{q}_b$ denote the coordinates of a point in body frame $B$. Then its coordinates in inertial frame $A$ are given by $\mathbf{q}_a = R_{ab}\mathbf{q}_b$. Multiple rotations compose via matrix multiplication: $R_{ac} = R_{ab}R_{bc}$ [1], reflecting the group structure of $SO(3)$.

A fundamental result in rotation theory, known as Euler's theorem, states that any rotation in $\mathbb{R}^3$ can be represented as a rotation about a fixed axis by a certain angle [1]. While rotation matrices provide a complete description of orientation, they require nine parameters to represent only three degrees of freedom. Unit quaternions offer a more compact and numerically stable encoding.

### B. Quaternion Algebra

Quaternion extended the concept of complex numbers to four dimensions, that is why we will review complex algebra, from which quaternions are naturally generalized.

*1) Complex Algebra:* A complex number takes the form

$$z = a + ib$$

where

$$\begin{gathered} a \text{ is the real part,} \\ b \text{ is the imaginary part, and} \\ i = \sqrt{-1} \end{gathered}$$

for example

$$\begin{gathered} z = 5 + 6i, \\ z = 6 - 13i, \\ \text{etc} \end{gathered}$$

Either part may be zero, which implies that the set of real numbers $\mathbb{R}$ is a subset of complex numbers $\mathbb{C}$ [2], [3].

The conjugate of a complex number $z = a + ib$ is defined as

$$z^* = a - ib$$

Complex multiplication follows the rule $i^2 = -1$. Given two complex numbers $z_1 = a_1 + ib_1$ and $z_2 = a_2 + ib_2$, their product is

$$z_1 z_2 = (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + a_2 b_1)$$

The magnitude (or modulus) of a complex number is given by

$$|z| = \sqrt{a^2 + b^2} = \sqrt{zz^*}$$

A unit complex number satisfies $|z| = 1$, which can be expressed in polar form as $z = \cos\theta + i\sin\theta = e^{i\theta}$, representing a rotation by angle $\theta$ in the two-dimensional plane.

*2) Introduction to Quaternion:* Quaternions were introduced by Sir William Rowan Hamilton as an extension of complex numbers from the two-dimensional plane $\mathbb{R}^2$ to a four-dimensional algebra over $\mathbb{R}$ [4], [2]. Knowing that a complex number in $\mathbb{R}^2$ has the form

$$z = a + ib$$

it is reasonable to presume that a complex number in $\mathbb{R}^3$ should take the form

$$z = a + ib + jc$$

where $i$ and $j$ are unit imaginaries: $i^2 = j^2 = -1$. However, when two such objects are multiplied together we have

$$z_1 z_2 = (a_1 + ib_1 + jc_1)(a_2 + ib_2 + jc_2)$$

which expands to

$$\begin{aligned} z_1 z_2 = {} & (a_1 a_2 - b_1 b_2 - c_1 c_2) \\ & + i(a_1 b_2 + b_1 a_2) + j(a_1 c_2 + c_1 a_2) + ijb_1 c_2 + jic_1 b_2 \end{aligned}$$

leaving the terms $ij$ and $ji$ undefined [2], [5].

Hamilton then extended the triple to a 4-tuple:

$$z = a + ib + jc + kd.$$

When two such objects are multiplied together, while substituting $i^2 = j^2 = k^2 = -1$ and using the multiplication rules

$$ij = k \quad jk = i \quad ki = j \quad ji = -k \quad kj = -i \quad ik = -j$$

we have

$$z_1 z_2 = (a_1 + ib_1 + jc_1 + kd_1)(a_2 + ib_2 + jc_2 + kd_2)$$

simplifies to

$$\begin{aligned} z_1 z_2 = {} & a_1 a_2 - (b_1 b_2 + c_1 c_2 + d_1 d_2) \\ & + a_1(ib_2 + jc_2 + kd_2) + a_2(ib_1 + jc_1 + kd_1) \\ & + i(c_1 d_2 - d_1 c_2) + j(d_1 b_2 - b_1 d_2) + k(b_1 c_2 - c_1 b_2) \end{aligned}$$

which shows that the product of two quaternions is again a quaternion [4], [2].

A quaternion $q = a + ib + jc + kd$ can be written as

$$q = (a, \mathbf{v}),$$

where $a \in \mathbb{R}$ is called the scalar part and $\mathbf{v} = (b, c, d) \in \mathbb{R}^3$ is called the vector part [2], [4]. The conjugate of $q$ is defined as

$$q^* = a - ib - jc - kd,$$

and the norm of $q$ is given by

$$\|q\| = \sqrt{a^2 + b^2 + c^2 + d^2} = \sqrt{qq^*}[4].$$

Quaternions with unit norm, $\|q\| = 1$, are called unit quaternions and form the basis for representing three-dimensional rotations in subsequent sections [4], [2].

*3) Quaternion Rotations and SO(3):* A three-dimensional vector $\mathbf{v} = (v_x, v_y, v_z)$ can be represented as a *pure quaternion*

$$p = 0 + v_x i + v_y j + v_z k,$$

that is, a quaternion whose scalar part is zero. In this way, the usual vectors in $\mathbb{R}^3$ can be viewed as a special subset of the quaternions [4].

To describe rotations, unit quaternions are used, any unit quaternion can be written in the form

$$q = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\,(u_x i + u_y j + u_z k),$$

where $\theta$ is a rotation angle and $\mathbf{u} = (u_x, u_y, u_z)$ is a unit vector giving the axis of rotation [4]. This shows that a three-dimensional rotation can be encoded by one axis and one angle.

If a vector $\mathbf{v}$ is represented as the pure quaternion $p$, the rotated vector $\mathbf{v}'$ is obtained by the operation

$$v' = qvq^{-1},$$

where $q$ is a unit quaternion and $q^{-1}$ is its inverse [4], [6]. The imaginary part of $p'$ gives the new vector $\mathbf{v}'$, and the length of the vector is preserved, so this operation defines a genuine rotation in $\mathbb{R}^3$.

This quaternion-based rotation can also be written using a $3 \times 3$ rotation matrix. For every unit quaternion $q$, there is an associated rotation matrix $R(q)$ such that

$$\mathbf{v}' = R(q)\,\mathbf{v},$$

and this matrix is always orthogonal with determinant one, hence it lies in $SO(3)$ [4]. Unit quaternions therefore provide a compact and singularity-free way to represent three-dimensional rotations, while composition of rotations corresponds simply to quaternion multiplication.

## III. Methods

### A. Scope and Limitations

The author decided to only consider rotational motion of rigid bodies in three-dimensional Euclidean space $\mathbb{R}^3$. Translational motion, scaling, external disturbances, and dynamic effects are not included. All rotations are specified using axis–angle parameters, and all computations are carried out using double-precision floating-point arithmetic.

The main objective is to evaluate unit quaternions as a mathematical representation of rigid-body orientation. Rotation matrices are included only as a reference baseline, in order to quantify numerical stability and computational efficiency behavior relative to unit quaternions.

### B. Experimental Setup

All experiments are performed under a common setup to ensure a fair comparison between unit quaternions and rotation matrices.

- Computational Environment: All simulations are implemented in Python 3 using standard numerical linear algebra routines for vector and matrix operations.
- Rotation Definition: Each rotation is specified by an axis–angle pair $(\mathbf{u}, \theta)$, where $\mathbf{u} \in \mathbb{R}^3$ is a unit vector representing the rotation axis and $\theta$ is the rotation angle.
- Input: The same axis–angle data are used to construct both rotation matrices and unit quaternions, ensuring that both representations describe the same geometric rotation.
- Rotation Matrices: For each axis–angle pair, a rotation matrix $R \in SO(3)$ is constructed and applied to a vector $\mathbf{v}$ using $\mathbf{v}' = R\mathbf{v}$. Composition of rotations is performed through matrix multiplication.
- Unit Quaternions: From the same axis–angle data, a unit quaternion
$$q = \left(\cos\frac{\theta}{2}, \sin\frac{\theta}{2}\,\mathbf{u}\right)$$
is constructed. Vectors are embedded as pure quaternions and rotated using $p' = qpq^{-1}$ formula. Multiple rotations are composed by quaternion multiplication, with optional normalization to maintain unit length.

Based on this setup, two experiments are conducted to examine numerical stability and computational efficiency.

### C. Numerical Stability Under Repeated Rotations

This experiment examines how numerical errors accumulate when the same rotation is applied repeatedly. A single rotation is defined using an axis–angle pair and represented both as a rotation matrix and as a unit quaternion. An initial vector in $\mathbb{R}^3$ is then rotated many times using each representation.

For the quaternion-based method, the vector is updated using repeated applications of $v' = qvq^{-1}$ formula. For the matrix-based method, the same rotation is applied using repeated matrix vector multiplication, with an accumulated rotation matrix updated through matrix multiplication.

After a large number of iterations, the deviation of the final vector from its expected direction is measured. In addition, for rotation matrices, the loss of orthogonality can be evaluated by measuring how far the accumulated matrix deviates from $SO(3)$. These measurements provide a direct comparison of numerical stability between the two representations.

### D. Computational Cost of Rotation Composition

This experiment compares the computational effort required to compose multiple rotations into a single rotation. A sequence of rotations is specified using axis–angle data and converted into both unit quaternions and rotation matrices.

For unit quaternions, composition is performed by successive quaternion multiplications starting from the identity quaternion. For rotation matrices, composition is performed by successive multiplication of $3 \times 3$ matrices starting from the identity matrix.

The execution time required to compute the final composed rotation is recorded for both methods using the same number of rotations. This comparison highlights the difference in computational cost arising between unit quaternions and rotation matrix representation.

## IV. Implementation

### A. Quaternion Class Declaration

```python
class Quaternion:

    def __init__(self, w, x, y, z):
        """Initialize quaternion q = w + xi + yj + zk"""
        self.w = w
        self.x = x
        self.y = y
        self.z = z

    @staticmethod
    def from_axis_angle(axis, angle):
        """Bangun unit quaternion dari axis-angle"""
        axis = np.array(axis)
        axis = axis / np.linalg.norm(axis)
        half_angle = angle / 2
        w = np.cos(half_angle)
        xyz = np.sin(half_angle) * axis
        return Quaternion(w, xyz[0], xyz[1], xyz[2])

    def conjugate(self):
        """Return conjugate q* = w - xi - yj - zk"""
        return Quaternion(self.w, -self.x, -self.y, -self.z)

    def norm(self):
        """Return norm ||q||"""
        return np.sqrt(self.w**2 + self.x**2 + self.y**2 + self.z**2)

    def normalize(self):
        """Normalize quaternion terhadap unit length"""
        n = self.norm()
        self.w /= n
        self.x /= n
        self.y /= n
        self.z /= n

    def multiply(self, other):
        """Quaternion multiplication: self * other"""
        w = self.w * other.w - self.x * other.x - self.y * other.y - self.z * other.z
        x = self.w * other.x + self.x * other.w + self.y * other.z - self.z * other.y
        y = self.w * other.y - self.x * other.z + self.y * other.w + self.z * other.x
        z = self.w * other.z + self.x * other.y - self.y * other.x + self.z * other.w
        return Quaternion(w, x, y, z)

    def rotate_vector(self, v):
        """Rotate vector v menggunakan rumus q*v*q^(-1)"""

        p = Quaternion(0, v[0], v[1], v[2])

        q_conj = self.conjugate()
        result = self.multiply(p).multiply(q_conj)

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

Fig. 2: Quaternion Class Declaration

*source: Author's archive*

The code shown in Fig. 2 implements a `Quaternion` class used to represent rigid-body orientation in $\mathbb{R}^3$. The class stores the scalar and vector components of a quaternion and provides essential operations such as quaternion multiplication, conjugation, normalization, vector rotation, etc.

### B. Rotation Matrix Declaration

```python
def rotation_matrix_from_axis_angle(axis, angle):
    """Bangun 3x3 rotation matrix dari axis-angle"""
    axis = np.array(axis)
    axis = axis / np.linalg.norm(axis)

    K = np.array([
        [0, -axis[2], axis[1]],
        [axis[2], 0, -axis[0]],
        [-axis[1], axis[0], 0]
    ])

    R = np.eye(3) + np.sin(angle) * K + (1 - np.cos(angle)) * (K @ K)
    return R

def check_orthogonality(R):
    """Mengecek deviation dari orthogonality: ||R^T R - I||"""
    deviation = np.linalg.norm(R.T @ R - np.eye(3))
    return deviation
```

Fig. 3: Rotation Matrix Builder Function

*source: Author's archive*

The code shown in Fig. 3 implements a function for constructing a $3 \times 3$ rotation matrix from an axis–angle. The resulting matrix belongs to the special orthogonal group $SO(3)$ and represents the same geometric rotation as the corresponding unit quaternion. This matrix-based implementation is included as a baseline for comparison with the quaternion-based approach.

### C. Numerical Stability Test

```python
def experiment_numerical_stability(test_case):
    """Membandingkan numerical stability antara quaternions dan rotation matrix"""
    print("=" * 70)
    print("NUMERICAL STABILITY UNDER REPEATED ROTATIONS")
    print("=" * 70)

    axis = np.array(test_case['axis'])
    angle = np.radians(test_case['angle_deg'])
    n_iterations = test_case['n_iterations']

    # Initial vector
    v0 = np.array([1.0, 0.0, 0.0])

    # Bangun quaternion dan rotation matrix
    q = Quaternion.from_axis_angle(axis, angle)
    R = rotation_matrix_from_axis_angle(axis, angle)

    # compute bagian quaternion
    v_quat = v0.copy()
    for i in range(n_iterations):
        v_quat = q.rotate_vector(v_quat)
        if i % 500 == 0:
            q.normalize()

    # compute bagian rotation matrix
    v_mat = v0.copy()
    R_accumulated = R.copy()
    for i in range(n_iterations):
        v_mat = R @ v_mat
        if i < n_iterations - 1:
            R_accumulated = R @ R_accumulated

    # expected result
    total_angle = n_iterations * angle
    R_expected = rotation_matrix_from_axis_angle(axis, total_angle)
    v_expected = R_expected @ v0

    # Compute errors
    error_quat = np.linalg.norm(v_quat - v_expected)
    error_mat = np.linalg.norm(v_mat - v_expected)
    orthogonality_error = check_orthogonality(R_accumulated)
```

Fig. 4: Numerical Stability Function

*source: Author's archive*

The code shown in Fig. 4 implements the numerical stability experiment described in the Methods section. A fixed axis–angle rotation is applied repeatedly to an initial vector using

both unit quaternions and rotation matrices. The experiment evaluates the accumulation of numerical errors by comparing the final rotated vectors with the expected theoretical result, as well as by measuring the loss of orthogonality in the accumulated rotation matrix.

### D. Computation Efficiency Test

```python
def experiment_computational_cost(test_case):
    """Membandingkan execution time"""
    n_rotations = test_case['n_rotations_cost']

    print("\n" + "=" * 70)
    print("COMPUTATIONAL COST OF ROTATION COMPOSITION")
    print("=" * 70)

    # Generate random rotations
    np.random.seed(42)
    axes = [np.random.randn(3) for _ in range(n_rotations)]
    angles = [np.random.uniform(0, 2 * np.pi) for _ in range(n_rotatio

    # compute quaternion
    start = perf_counter()
    q_result = Quaternion(1, 0, 0, 0)
    for axis, angle in zip(axes, angles):
        q = Quaternion.from_axis_angle(axis, angle)
        q_result = q_result.multiply(q)
    time_quat = perf_counter() - start

    # compute rotation matrix
    start = perf_counter()
    R_result = np.eye(3)
    for axis, angle in zip(axes, angles):
        R = rotation_matrix_from_axis_angle(axis, angle)
        R_result = R_result @ R
    time_mat = perf_counter() - start
```

Fig. 5: Computation Efficiency Function

*source: Author's archive*

The code shown in Fig. 5 implements the computational efficiency experiment for rotation composition. A sequence of rotations is composed using both quaternion multiplication and matrix multiplication, and the execution time of each method is recorded.

The full implementation code used in this section can be reviewed in the appendix.

## V. Experiment

### A. Test Cases

The author has designed three test cases to evaluate the behavior of unit quaternions under different rotational conditions. Each test case specifies a rotation axis, a rotation angle, and parameters controlling the number of repeated iteration or rotation.

For each test case, identical axis–angle parameters are used to construct both the unit quaternion representation and the corresponding rotation matrix. This ensures that any observed differences in numerical stability or computational efficiency arise solely from the underlying mathematical representation.

```python
TEST_CASES = {
    # Test Case 1: Rotasi kecil, banyak iterasi
    1: {
        'name': 'Small angle, many iterations',
        'axis': [1.0, 2.0, 3.0],
        'angle_deg': 5.0,
        'n_iterations': 50000,
        'n_rotations_cost': 1000
    },
    # Test Case 2: Rotasi besar, iterasi sedang
    2: {
        'name': 'Large angle, medium iterations',
        'axis': [0.0, 1.0, 0.0],
        'angle_deg': 45.0,
        'n_iterations': 10000,
        'n_rotations_cost': 5000
    },
    # Test Case 3: Rotasi arbitrary axis, iterasi sangat banyak
    3: {
        'name': 'Arbitrary axis, stress test',
        'axis': [1.0, 1.0, 1.0],
        'angle_deg': 1.0,
        'n_iterations': 100000,
        'n_rotations_cost': 10000
    }
}
```

Fig. 6: Test Cases

*source: Author's archive*

### B. Test Case 1

In this test case, a small rotation of $5^\circ$ is applied repeatedly for 50,000 iterations about a normalized axis $(1, 2, 3)$. The results show that both unit quaternions and rotation matrices produce identical final vector orientations, matching the expected rotation outcome with numerical errors on the order of $10^{-12}$ (see Fig. 7). This indicates that both representations exhibit strong numerical stability under repeated rotation. Although the rotation matrix approach yields a slightly smaller final error, the difference is not practically significant, and the loss of orthogonality remains negligible. In terms of computational performance, rotation composition using unit quaternions is approximately $1.81\times$ faster than using rotation matrices, highlighting the efficiency advantage of unit quaternions for repeated rigid-body orientation updates.

```
======================================================================
TEST CASE 1: Small angle, many iterations
======================================================================

======================================================================
NUMERICAL STABILITY UNDER REPEATED ROTATIONS
======================================================================

Rotasi: 5.0° dengan axis [0.26726124 0.53452248 0.80178373]
Jumlah rotasi: 50000

------------------------------------------------------------
VECTOR POSITIONS:
------------------------------------------------------------
  Initial:    [  1.00000000,   0.00000000,   0.00000000]
  Quaternion: [ -0.80114315,   0.55132513,   0.23283096]
  Matrix:     [ -0.80114315,   0.55132513,   0.23283096]
  Expected:   [ -0.80114315,   0.55132513,   0.23283096]

Error di final vector position:
  Quaternion method:      3.58e-12
  Rotation matrix method: 4.11e-13
  Accuracy ratio:         0.11x

Orthogonality deviation:
  Rotation matrix method: 1.51e-12

======================================================================
COMPUTATIONAL COST OF ROTATION COMPOSITION
======================================================================

Jumlah rotasi: 1000

Execution time:
  Quaternion method: 2.91 ms
  Matrix method:     5.27 ms
  Speedup factor:    1.81x

======================================================================
SUMMARY OF RESULTS
======================================================================

1. Numerical Stability:
   Kedua metode memiliki akurasi sebanding di level ~4e-12
   Kehilangan orthogonalitas matriks: 1.51e-12

2. Computational Efficiency:
   Quaternions 1.81x lebih cepat
```

Fig. 7: Test Case 1 Output

*source: Author's archive*

### C. Test Case 2

In this test case, a larger rotation of $45^\circ$ about the $y$-axis is applied repeatedly for 10,000 iterations. The results show that both unit quaternions and rotation matrices produce identical final vector orientations, matching the expected rotation outcome with numerical errors on the order of $10^{-13}$ (see Fig. 8). Unit quaternion approach exhibits a lower final error than the rotation matrix method. The orthogonality deviation of the accumulated rotation matrix remains negligible. In terms of performance, unit quaternions again outperform rotation matrices, achieving approximately a $1.8\times$ speedup in rotation composition.

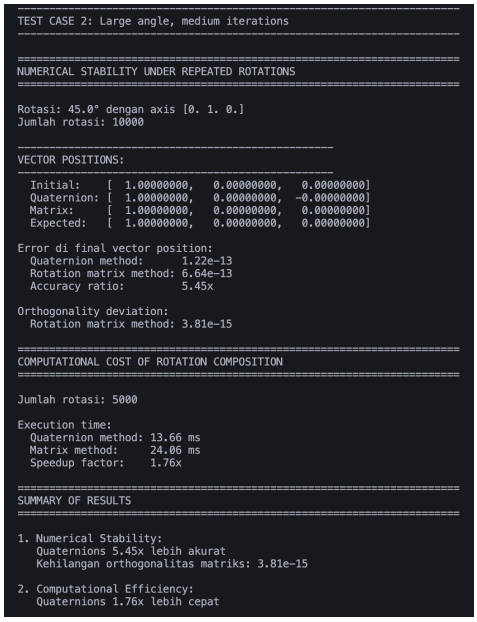


Fig. 8: Test Case 2 Output

*source: Author's archive*

### D. Test Case 3

In this test case, a small rotation of $1^\circ$ about an arbitrary axis $(1, 1, 1)$ is applied repeatedly for 100,000 iterations. Despite the large number of repeated rotations, both unit quaternions and rotation matrices produce identical final vector orientations, matching the expected rotation outcome with numerical errors on the order of $10^{-12}$ (see Fig. 9). This demonstrates that both representations maintain good numerical stability even under heavy iterative composition. However, the accumulated rotation matrix exhibits a larger loss of orthogonality compared to previous test cases. In terms of computational efficiency, unit quaternions remain consistently faster, achieving approximately a $1.76\times$ speedup over rotation matrices.

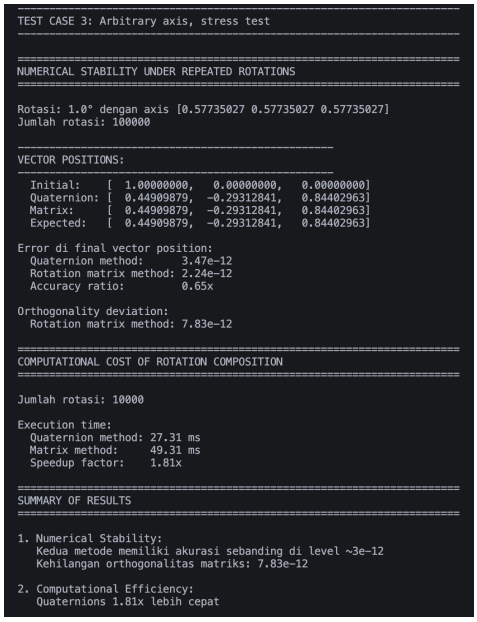


Fig. 9: Test Case 3 Output

*source: Author's archive*

## VI. Conclusion

This paper has shown the representation of rigid-body orientation in three-dimensional space using unit quaternions. An implementation was developed to apply and compose rotations using unit quaternions, with rotation matrices included solely as a reference representation to support empirical validation.

The experimental results demonstrate that unit quaternions provide a numerically stable representation of rigid-body orientation under repeated rotation and composition. The ability to maintain a valid orientation through simple normalization further reinforces the suitability of unit quaternions for long sequences of rotational operations. In addition, the experiments show that quaternion-based rotation composition can be performed more efficiently than matrix-based composition, reflecting the reduced parameterization and simpler algebraic structure of unit quaternions.

These findings confirm that unit quaternions give an effective and practical representation for rigid-body orientation in $\mathbb{R}^3$. By combining numerical robustness, computational efficiency, and conceptual clarity, unit quaternions are well suited for applications that require reliable orientation handling, such as computer graphics, robotics, and simulation, where stable and efficient rotational representations are essential.

## VII. Appendix

- Source Code : *https://github.com/Valz0504/Makalah-IF2123-rigid-body-orientation-using-unit-quaternions*
- Video : *https://youtu.be/EV2VQNi-KaA*

## VIII. Acknowledgement

The author would like to express sincere gratitude to Dr. Ir. Rinaldi, M.T., lecturer of the IF2123 Linear Algebra and Geometry course, for his guidance, encouragement, and

support throughout the semester. Thanks to his support, the author was able to gain valuable insights and learn many new things throughout the course. Next, the author would thank to family and friends who have been keep motivating and supporting. Finally, the author acknowledges the use of artificial intelligence tools to assist in language refinement and test case creation.

## REFERENCES

[1] R. M. Murray, Z. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*. Boca Raton, Florida: CRC Press, 1994, [Accessed: 20-Dec-2025]. [Online]. Available: https://www.cse.lehigh.edu/~trink/Courses/RoboticsII/reading/murray-li-sastry-94-complete.pdf

[2] J. Vince, *Geometric Algebra for Computer Graphics*. London: Springer-Verlag London, 2008, [Accessed: 20-Dec-2025]. [Online]. Available: https://dokumen.pub/geometric-algebra-for-computer-graphics-1nbsped-1846289963-9781846289965.html

[3] R. Munir, "Aljabar kompleks," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-25-Aljabar-Kompleks-2025.pdf, 2025, [Accessed: 20-Dec-2025].

[4] J. Gallier and J. Quaintance, *Linear Algebra for Computer Vision, Robotics, and Machine Learning*. Philadelphia, PA: University of Pennsylvania, 2025, version July 25, 2025. [Online]. Available: https://www.cis.upenn.edu/~cis5150/linalg-I-f.pdf

[5] R. Munir, "Aljabar quaternion (bagian 1)," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-26-Aljabar-Quaternion-Bagian1-2025.pdf, 2025, [Accessed: 20-Dec-2025].

[6] ——, "Aljabar quaternion (bagian 2)," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2025-2026/Algeo-27-Aljabar-Quaternion-Bagian2-2025.pdf, 2025, [Accessed: 20-Dec-2025].

[7] H. Anton and C. Rorres, *Elementary Linear Algebra: Applications Version*, 12th ed. Hoboken, New Jersey: John Wiley & Sons, 2019, [Accessed: 20-Dec-2025]. [Online]. Available: https://www.studyhalo.com/media/resources/resources/MAT1503/Textbook/MAT1503_-_Prescribed_book.pdf

## STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 24 December 2025

Emilio Justin
13524043