Analysis of Rotation Methods in Computer Graphics

Yonatan Edward Njoto - 13523036[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523036@std.stei.itb.ac.id, [2]yonatan.njoto@gmail.com

***Abstract*— This paper explores the fundamental techniques used for 3D object rotation—Euler angles, quaternions, transformation matrices, and Rodrigues' formula—offering a comprehensive comparison of their strengths and limitations. The objective is to provide a detailed analysis of each method's execution speed, memory consumption, and computational efficiency, highlighting their impact on real-time applications, simulations, and graphics rendering.**

***Keywords*—Computer graphics, Rotations, Geometric transformations, Rodrigues, Quaternion, Euler Angles.**

I. Introduction

Rotations in three-dimensional space are a cornerstone of computer graphics, robotics, aerospace engineering, and physics simulations. The ability to manipulate objects in 3D environments with precision and efficiency is crucial for rendering realistic models and animations. Several mathematical techniques exist to represent and compute 3D rotations, each providing unique benefits and limitations. Among the most used methods are Euler angles, rotation matrices, quaternions, and Rodrigues' rotation formula.

Euler angles provide an intuitive way to describe rotations by specifying angles around coordinate axes. However, they are prone to gimbal lock, a condition that results in the loss of one degree of freedom. Matrix transformations, on the other hand, offer a straightforward and computationally efficient approach but can suffer from numerical instability if not properly normalized. Quaternions are widely regarded for their ability to represent rotations compactly and avoid gimbal lock, making them popular in modern graphics applications. Rodrigues' rotation formula provides an elegant way to compute rotations around arbitrary axes, offering efficiency and simplicity for small-angle rotations.

This paper aims to compare these methods by analyzing their mathematical properties, computational costs, and practical applications. By providing a comprehensive overview, we hope to highlight the strengths of each approach, assisting developers and researchers in selecting the most suitable method for their specific needs.

II. Theoretical Background

A. Euler Angles

Euler Angles are a set of three angles used to describe the orientation of a rigid body in a three-dimensional space. The concept was introduced by the Swiss mathematician Leonhard Euler in the 18th century. The angles define rotations around a fixed coordinate system (typically denoted as x, y, z) in a specific sequence. Euler angles are particularly useful in applications like 3D graphics, robotics, and aerospace engineering, where rotations need to be described in terms of a series of simple, successive rotations [1].

Euler angles consist of three rotations, typically denoted as:

a. Pitch (θ): Rotation about the x-axis.

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

b. Yaw (ψ): Rotation about the y-axis.

$$R_y(\psi) = \begin{bmatrix} cos(\psi) & 0 & sin(\psi) \\ 0 & 1 & 0 \\ -sin(\psi) & 0 & \cos(\psi) \end{bmatrix} \quad (2)$$

c. Roll (ϕ): Rotation about the z-axis.

$$R_z(\phi) = \begin{bmatrix} cos(\phi) & -sin(\phi) & 0 \\ sin(\phi) & cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3)$$

The total rotation matrix R is obtained by multiplying the individual rotation matrices in a specific order, depending on the rotation. For the ZYX convention (which is common), the final rotation matrix is:

$$R = R_z(\phi)R_y(\psi)R_x(\theta) \quad (4)$$

While Euler angles are widely used and intuitive, they come with several limitations:

a. Gimbal lock occurs when two of the three rotation axes align, causing a loss of one degree of freedom.
b. Ambiguity, Different sequences of rotations (e.g., XYZ vs. ZYX) can result in different final orientations.

B. Quaternions

Quaternions are a mathematical tool used to represent and compute rotations in three-dimensional space. Introduced by William Rowan Hamilton in 1843, quaternions extend complex numbers to higher dimensions, providing a compact and efficient way to describe 3D rotations without the limitations associated with Euler angles [2].

A quaternion is defined as:

$$q = w + xi + yj + zk \quad (5)$$

Where w, x, y, z are real numbers, and i, j, k are imaginary units that follow the rules:

$$i^2 = j^2 = k^2 = ijk = -1 \quad (6)$$

In the context of 3D graphics, quaternions are commonly written as:

$$q = (w, \vec{v}) = (w, x, y, z) \quad (7)$$

Where w is the scalar part, and $\vec{v} = (x, y, z)$ is the vector part. To rotate a vector $\vec{p}$ by a quaternion q, the vector is treated as a pure quaternion $p = (0, \vec{p})$. The rotation is applied by:

$$p' = qpq^{-1} \quad (8)$$

Where q^{-1} is the inverse of q. Advantages of using Quaternions compared to Euler Angles are no Gimbal Locks and efficient interpolation because it allows spherical linear interpolation (SLERP).

C. Rodrigues Rotation

Rodrigues' rotation formula provides a method to rotate a vector in three-dimensional space around an arbitrary axis by a given angle [3]. The formula is an efficient way to compute the rotation without needing to construct a full rotation matrix.

The formula is expressed as:

$$v_{rot} = v\cos(\theta) + (k \times v)\sin(\theta) + k(k.v)(1 - \cos(\theta)) \quad (9)$$

Where:

a. v is the vector to be rotated
b. k is the unit vector along the axis of rotation
c. θ is the angle of rotation
d. v_{rot} is the rotated vector

III. Implementation

This implementation visualizes the rotation of a cube using three different methods:

a. Euler Rotator
b. Transformation Matrix Rotator
c. Quaternion Rotator
d. Rodrigues Rotator

The cube is rotated by the same Euler angles (degrees in ZYX order) across all methods. The goal is to compare the results visually and its operation cost, highlight differences between the three techniques.

The implementation is in Python, leveraging efficient array processing for matrix operations, dot products, and transformations, ensuring optimal performance during visualization.

A. Cube

The Cube class generates vertices representing a cube centered at the origin. The create cube method defines the eight vertices based on the cube's size.

```python
class Cube:
    def __init__(self, size=1):
        self.size = size
        self.vertices = self._create_cube()

    def _create_cube(self):
        s = self.size / 2
        return np.array([
            [s, s, s], [s, s, -s], [s, -s, s], [s, -s, -s],
            [-s, s, s], [-s, s, -s], [-s, -s, s], [-s, -s, -s]
        ])
```

Figure 1. Cube Class
Source: https://github.com/yonatan-nyo/3d-rotations

B. Plotter

Plotter is responsible for drawing the cube in 3D space using Matplotlib. Edges are plotted by connecting vertex pairs to form the cube structure.

```python
class Plotter:
    @staticmethod
    def plot_cube(ax, vertices, color='b'):
        edges = [
            (0, 1), (1, 3), (3, 2), (2, 0),  # Top edges
            (4, 5), (5, 7), (7, 6), (6, 4),  # Bottom edges
            (0, 4), (1, 5), (2, 6), (3, 7)  # Vertical edges
        ]
        for edge in edges:
            points = vertices[np.array(edge)]
            ax.plot3D(*points.T, color)
```

Figure 2. Plotter Class
Source: https://github.com/yonatan-nyo/3d-rotations

C. Operation Converter

To simplify the comparison of different rotation methods, we will first convert Euler angles into quaternions. By converting Euler angles to quaternions, we can directly compare the performance of quaternions against other methods easily, while maintaining consistency in how rotations are applied. The process involves representing the Euler angles as a combination of rotations around the three principal axes (x, y, and z), and then converting this representation into a quaternion form, then passing them into the respective functions to apply rotation into the cube vertices. Below is a visual representation of the conversion process:

```python
def euler_to_quaternion(roll, pitch, yaw, order='XYZ'):
    r = R.from_euler(order.lower(), [roll, pitch, yaw], degrees=True)
    return r.as_quat()
```

Figure 3. Euler to Quaternion
Source: https://github.com/yonatan-nyo/3d-rotations

Additionally, to facilitate Rodrigues rotation, we will convert Euler angles into the axis-angle representation. The axis-angle representation encodes a rotation as a unit vector (representing

the axis of rotation) and a scalar (representing the angle of rotation). By converting Euler angles to this form, we can perform rotations efficiently using geometric algebra principles. The process for this conversion is illustrated in the diagram below:

```python
def euler_to_axis_angle(roll, pitch, yaw, order='XYZ'):
    r = R.from_euler(order.lower(), [roll, pitch, yaw], degrees=True)
    angle = np.linalg.norm(r.as_rotvec())
    axis = r.as_rotvec() / (angle if angle > 1e-6 else 1)
    return angle, axis
```

Figure 4. Euler to Axis Angle
Source: https://github.com/yonatan-nyo/3d-rotations

D. Euler Rotator

The Euler Rotator performs rotations using Euler angles, applying them through both SciPy functions and manual calculations. This method involves rotating objects around the three principal axes (x, y, and z), with each axis defined by a corresponding angle that specifies the object's orientation in 3D space. While SciPy provides optimized functions for handling these rotations efficiently, manual calculations allow for a deeper understanding of the underlying mathematical principles and the rotational mechanics involved.

In addition to full 3D rotations, the Euler Rotator also supports single-axis rotations, which significantly simplifies the process. For single-axis rotations, only one of the three principal axes (x, y, or z) is involved, making the rotation computationally simpler and faster. This is particularly useful for cases where the rotation is confined to a single axis, as it avoids the complexity of applying transformations along multiple axes simultaneously.

By implementing both SciPy-based and manual approaches, the Euler Rotator provides flexibility in handling various rotation scenarios, allowing for a direct comparison of the performance and efficiency of both methods. This comparison offers insights into the trade-offs between convenience (via SciPy) and control (via manual calculations), while also highlighting the practical benefits of Euler angles for different types of rotations.

```python
class Rotator:
    @staticmethod
    def euler_one_degree(vertices, angles, degree: Literal['X', 'Y', 'Z']):
        if (degree == 'X'):
            rx = np.array([
                [1, 0, 0],
                [0, np.cos(np.radians(angles)), -np.sin(np.radians(angles))],
                [0, np.sin(np.radians(angles)), np.cos(np.radians(angles))]
            ])
            return np.dot(vertices, rx.T)
        elif (degree == 'Y'):
            ry = np.array([
                [np.cos(np.radians(angles)), 0, np.sin(np.radians(angles))],
                [0, 1, 0],
                [-np.sin(np.radians(angles)), 0, np.cos(np.radians(angles))]
            ])
            return np.dot(vertices, ry.T)
        elif (degree == 'Z'):
            rz = np.array([
                [np.cos(np.radians(angles)), -np.sin(np.radians(angles)), 0],
                [np.sin(np.radians(angles)), np.cos(np.radians(angles)), 0],
                [0, 0, 1]
            ])
            return np.dot(vertices, rz.T)
        else:
            return vertices

    @staticmethod
    def euler(vertices, angles, order='ZYX'):
        # Convert angles to radians if they are in degrees
        r = R.from_euler(order, angles, degrees=True)
        return r.apply(vertices)

    @staticmethod
    def euler_manual(vertices, angles, order='ZYX'):
        angles = np.radians(angles)
        cz = np.cos(angles[0])
        sz = np.sin(angles[0])
        cy = np.cos(angles[1])
        sy = np.sin(angles[1])
        cx = np.cos(angles[2])
        sx = np.sin(angles[2])

        rz = np.array([
            [cz, -sz, 0],
            [sz, cz, 0],
            [0, 0, 1]
        ])

        ry = np.array([
            [cy, 0, sy],
            [0, 1, 0],
            [-sy, 0, cy]
        ])

        rx = np.array([
            [1, 0, 0],
            [0, cx, -sx],
            [0, sx, cx]
        ])

        vertices = np.dot(vertices, rx.T)
        vertices = np.dot(vertices, ry.T)
        vertices = np.dot(vertices, rz.T)

        return vertices

    # other methods
```

Figure 5. Euler Rotation
Source: https://github.com/yonatan-nyo/3d-rotations

E. Transformation Matrix Rotator

The Matrix Rotator implements a 3D rotation transformation using a combination of rotation matrices, like the Euler rotation method. This approach involves constructing individual rotation matrices for each of the three principal axes (x, y, and z), and then multiplying them together to form a single composite matrix. This matrix is then used to rotate 3D coordinates. While this method provides a straightforward way to handle rotations, it can become computationally expensive when dealing with multiple transformations, making it less efficient than alternative methods like quaternions in some cases.

```python
class Rotator:
    # other methods

    @staticmethod
    def matrix(vertices, angles):
        # Convert angles to radians
        angles = np.radians(angles)

        # Create rotation matrices for each axis
        ry = np.array([
            [np.cos(angles[1]), 0, np.sin(angles[1])],
            [0, 1, 0],
            [-np.sin(angles[1]), 0, np.cos(angles[1])]
        ])

        rz = np.array([
            [np.cos(angles[0]), -np.sin(angles[0]), 0],
            [np.sin(angles[0]), np.cos(angles[0]), 0],
            [0, 0, 1]
        ])

        rx = np.array([
            [1, 0, 0],
            [0, np.cos(angles[2]), -np.sin(angles[2])],
            [0, np.sin(angles[2]), np.cos(angles[2])]
        ])

        # Compute combined rotation matrix
        rotation_matrix = rz @ ry @ rx

        return np.dot(vertices, rotation_matrix.T)

    # other methods
```

Figure 6. Matrix Rotator
Source: https://github.com/yonatan-nyo/3d-rotations

F. Quaternion Rotator

The Quaternions provide robust 3D rotation, effectively overcoming issues like gimbal lock that can occur with traditional Euler angles. By using quaternions, rotations can be represented in a compact form, reducing computational complexity and improving numerical stability. This implementation takes advantage of the optimized SciPy library in Python, which offers high-performance functions for quaternion-based rotations, ensuring both precision and speed. This makes quaternions an ideal choice for applications that require smooth and efficient 3D transformations, such as computer graphics, robotics, and physics simulations.

```python
class Rotator:
    # other methods

    @staticmethod
    def quaternion(vertices, quaternion):
        # Normalize the quaternion to ensure a valid rotation
        quaternion = np.array(quaternion)
        if np.linalg.norm(quaternion) != 1:
            quaternion = quaternion / np.linalg.norm(quaternion)
        rotation = R.from_quat(quaternion)
        return rotation.apply(vertices)

    # other methods
```

Figure 7. Quaternion Rotator
Source: https://github.com/yonatan-nyo/3d-rotations

G. Rodrigues Rotator

The Rodrigues rotation formula provides a method for rotating vectors in 3D space using an axis-angle representation. It simplifies the computation of rotation matrices by directly applying a rotation around an arbitrary axis, rather than using complex transformations for each of the three principal axes. By specifying the axis of rotation and the angle, the Rodrigues formula avoids the need for multiple matrix multiplications, offering a streamlined solution for performing rotations in 3D space.

```python
class Rotator:
    # other methods

    @staticmethod
    def rodrigues(vertices, axis, angle):
        # Normalize the axis of rotation
        axis = axis / np.linalg.norm(axis)
        cos_a = np.cos(angle)
        sin_a = np.sin(angle)
        dot = np.dot(vertices, axis)
        cross = np.cross(axis, vertices)
        # Apply Rodrigues' rotation formula to each vertex
        return vertices * cos_a + cross * sin_a + axis * dot[:, None] * (1 - cos_a)
```

Figure 8. Rodrigues Rotator
Source: https://github.com/yonatan-nyo/3d-rotations

IV. USAGE

To benchmark these five methods, we will conduct a thorough analysis of their performance by measuring both execution speed and memory usage. This will be done by utilizing Python's time module to track the time taken for execution, and trace malloc to monitor memory allocation throughout the process. Followed by analysis on how these methods work.

A. Single Axis Rotation

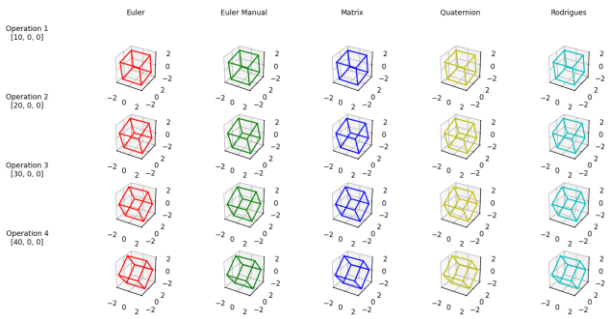

Figure 9. Single Axis Rotation
Source: https://github.com/yonatan-nyo/3d-rotations

In the case of single-axis rotation, Euler angles (one degree) offer the fastest method for performing the rotation because they simplify the transformation process. By rotating an object around a single axis, only one angle needs to be applied, eliminating the complexity of full 3D transformations. Euler angles directly specify the amount of rotation around the relevant axis, reducing both time and computational overhead. This makes them an efficient choice for rotations confined to a single axis.

Also, when considering peak memory usage, Euler angles (one degree) achieve the lowest peak memory consumption. This is because they only require a minimal amount of data—

just the single angle and axis to perform the rotation—without the need to create additional data structures like rotation matrices. In contrast, other methods such as quaternions, while offering lower total memory usage, may require extra steps like normalization or additional computations that lead to higher peak memory usage.

```
$ python main.py

rotations:  [[10, 0, 0], [20, 0, 0], [30, 0, 0], [40, 0, 0]]
Summary of Rotation Methods:
+-------------------------+--------------+-----------------+---------------+
| Method                  | Total Time   | Total Memory    | Peak Memory   |
+=========================+==============+=================+===============+
| euler_one_degree        | 0.000370s    | 0.001280MB      | 0.000584MB    |
+-------------------------+--------------+-----------------+---------------+
| euler_manual_all_degree | 0.000397s    | 0.001664MB      | 0.001568MB    |
+-------------------------+--------------+-----------------+---------------+
| matrix                  | 0.000836s    | 0.001664MB      | 0.001512MB    |
+-------------------------+--------------+-----------------+---------------+
| quaternion              | 0.000550s    | 0.001152MB      | 0.003563MB    |
+-------------------------+--------------+-----------------+---------------+
| rodrigues               | 0.001022s    | 0.002176MB      | 0.006912MB    |
+-------------------------+--------------+-----------------+---------------+

euler_one_degree has the lowest total time of 0.000370s
quaternion has the lowest total memory of 0.001152MB
```

Figure 10. Single Axis Rotation Result
Source: https://github.com/yonatan-nyo/3d-rotations

B. Two Axis Rotation

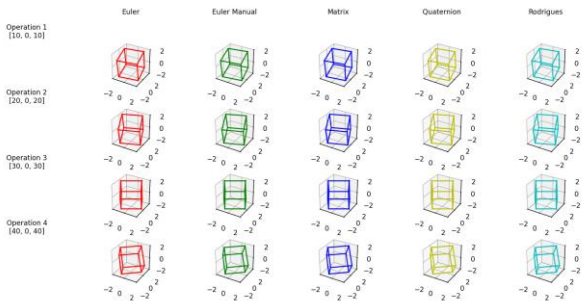

Figure 11. Two Axis Rotation
Source: https://github.com/yonatan-nyo/3d-rotations

In two-axis rotation, quaternions provide the best performance in terms of both execution time and total memory usage. This is due to their efficient representation of rotations, where a quaternion requires only four parameters to encode a rotation in 3D space, compared to the multiple values required for a rotation matrix. Additionally, quaternions avoid the need for intermediate computations, such as calculating sine and cosine values for multiple axes, which can lead to faster execution.

However, when it comes to peak memory usage, the transformation matrix performs better. A transformation matrix is typically a 3x3 matrix, and since it only requires the values for the rotation itself, there are no additional operations needed to normalize the rotation or transform values into axis components. In contrast, quaternions require normalization to ensure they represent valid rotations, which adds some extra memory overhead. Furthermore, quaternions involve handling additional data structures to store the rotational axis and angle, leading to slightly higher peak memory usage.

```
$ python main.py

rotations:  [[10, 0, 10], [20, 0, 20], [30, 0, 30], [40, 0, 40]]
Summary of Rotation Methods:
+-------------------------+--------------+-----------------+---------------+
| Method                  | Total Time   | Total Memory    | Peak Memory   |
+=========================+==============+=================+===============+
| euler                   | 0.000460s    | 0.001152MB      | 0.004539MB    |
+-------------------------+--------------+-----------------+---------------+
| euler_manual_all_degree | 0.000480s    | 0.001664MB      | 0.001568MB    |
+-------------------------+--------------+-----------------+---------------+
| matrix                  | 0.000595s    | 0.001664MB      | 0.001512MB    |
+-------------------------+--------------+-----------------+---------------+
| quaternion              | 0.000424s    | 0.001152MB      | 0.003563MB    |
+-------------------------+--------------+-----------------+---------------+
| rodrigues               | 0.000965s    | 0.002176MB      | 0.006912MB    |
+-------------------------+--------------+-----------------+---------------+

quaternion has the lowest total time of 0.000424s
quaternion has the lowest total memory of 0.001152MB
```

Figure 12. Two Axis Rotation Result
Source: https://github.com/yonatan-nyo/3d-rotations

C. Three Axis Rotation

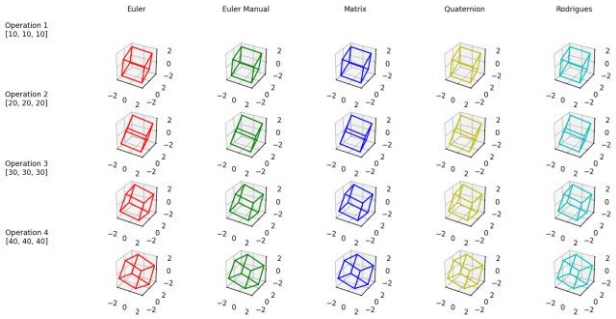

Figure 13. Three Axis Rotation
Source: https://github.com/yonatan-nyo/3d-rotations

In three-axis rotation, quaternions continue to deliver the best overall performance, in both execution time and total memory usage. This is largely due to the compact nature of quaternion representations, which encode 3D rotations using only four values (one scalar and three vector components). This efficiency minimizes the number of trigonometric calculations and matrix multiplications typically required by other rotation methods, resulting in faster computations and lower memory consumption over the duration of the operation.

However, when considering peak memory usage, the transformation matrix consistently outperforms quaternions. A 3x3 rotation matrix directly represents the rotation, requiring no additional steps such as normalization or axis-angle conversion. This direct representation eliminates the need to maintain extra data structures or perform iterative calculations during the rotation process, contributing to a lower memory footprint at peak usage. In contrast, quaternions require periodic normalization to prevent drift and ensure accurate rotation over time, which can momentarily increase memory usage.

Furthermore, in three-axis rotations, quaternions must often be converted from Euler angles or axis-angle representations, involving intermediate steps that add to memory demands. The transformation matrix, by contrast, applies rotations directly through matrix multiplication, bypassing the need for additional transformations and maintaining a lower peak memory requirement.

```
$ python main.py

rotations:  [[10, 10, 10], [20, 20, 20], [30, 30, 30], [40, 40, 40]]
Summary of Rotation Methods:
+-------------------------+-------------+---------------+--------------+
| Method                  | Total Time  | Total Memory  | Peak Memory  |
+=========================+=============+===============+==============+
| euler                   | 0.000498s   | 0.001152MB    | 0.004539MB   |
+-------------------------+-------------+---------------+--------------+
| euler_manual_all_degree | 0.000397s   | 0.001664MB    | 0.001568MB   |
+-------------------------+-------------+---------------+--------------+
| matrix                  | 0.000579s   | 0.001664MB    | 0.001512MB   |
+-------------------------+-------------+---------------+--------------+
| quaternion              | 0.000381s   | 0.001152MB    | 0.003563MB   |
+-------------------------+-------------+---------------+--------------+
| rodrigues               | 0.001006s   | 0.002112MB    | 0.006848MB   |
+-------------------------+-------------+---------------+--------------+

quaternion has the lowest total time of 0.000381s
quaternion has the lowest total memory of 0.001152MB
```

Figure 14. Three Axis Rotation Result
Source: https://github.com/yonatan-nyo/3d-rotations

V. Conclusion

The choice of rotation method depends on the application's needs. For single-axis rotations, Euler angles are the fastest and simplest option. For multi-axis rotations, quaternions provide the best performance and stability, avoiding issues like gimbal lock. Transformation matrices are useful when minimizing peak memory usage is a priority.

VI. Appendix

a. YouTube video explaining the concept: https://youtube.com/shorts/i2tTxdlhXh4
b. GitHub repository for the project: https://github.com/yonatan-nyo/3d-rotations

VII. Acknowledgment

The author would like to express sincere gratitude to God Almighty for the guidance and ease in writing this paper. Special thanks are also extended to Dr. Ir. Rinaldi Munir, M.T., for his role as the lecturer in the IF2123 Linear and Geometry Algebra course and for publishing the lecture materials on the website, which were instrumental in the research process. The author is deeply appreciative of the unwavering support from family and friends throughout the completion of this paper.

References

[1] Goldstein, H., Poole, C., & Safko, J. 2002. “Classical Mechanics (3rd ed.). Addison-Wesley”.
[2] Munir, Rinaldi. 2023. “Aljabar Quaternion (Bagian 2)”. https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2023-2024/Algeo-26-Aljabar-Quaternion-Bagian2-2023.pdf (Diakses pada 22 Desember 2024)
[3] Freitas, P. (2018). “Rodrigues' rotation formula: A historical and mathematical survey. International Congress of Mathematicians”. https://eta.impa.br/icm_files/short/section_19/SC.19.5.pdf (Diakses pada 30 Desember 2024)

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, 31 December 2024

Yonatan Edward Njoto
13523036