VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

Ho Chi Minh City University of Technology

Faculty of Applied Science

LINEAR ALGEBRA (MT1007)

Assignment

"Project 9: Orthogonal matrices and 3D graphics"

Instructor: PhD. Dau The Phiet – CSE–HCMUT

Semester: 242
Class: CC13
Group: 04
Students: Nguyen Tang Nhat Anh – 2452069
Cao Nguyen Hoang Duy – 2452190
Nguyen Hoang Quoc – 2353027
Phan Cam – 2252087
Le Pham Tien Long – 2352688
Truong Xuan Sang – 2353045
Nguyen Vu Long – 2352696

Ho Chi Minh City, June 2025

Member List and Workload

No.	Fullname	Student ID	Works	Distribution
1	Nguyen Tang Nhat Anh	2452069	Theoretical Foundation	14.28%
2	Cao Nguyen Hoang Duy	2452190	Conclusion and Further Discussion	14.28%
3	Nguyen Hoang Quoc	2353027	Problem Statement	14.28%
4	Phan Cam	2252087	Revise report	14.28%
5	Le Pham Tien Long	2352688	Code and LaTeX editing	14.28%
6	Truong Xuan Sang	2353045	Illustrative Example	14.28%
7	Nguyen Vu Long	2352696	Code and Revise report	14.28%

Contents

List of Figures

List of Listings

1 Problem Statement

Applying orthogonal matrices in three-dimensional (3D) graphics addresses the fundamental problem of accurately representing and manipulating three-dimensional objects within a two-dimensional (2D) viewing space, such as a computer screen. This problem is significant because precise rotations and transformations are crucial in fields of computer graphics and virtual reality. We need to generate a projection to show the object on the screen by simply discarding one coordinate. This provides a limited and often inadequate understanding of the object's true shape, as it restricts viewing to only a few static angles [1]. The central challenge involves applying mathematical concepts to ensure objects can be rotated and visualized consistently without distortion of their intrinsic geometrical properties.
Orthogonal matrices—square matrices with orthonormal columns and rows—provide a critical tool for achieving this stability, as their defining property

$$Q^TQ = QQ^T = I$$

yields the critical relationship $Q^T = Q^{-1}$, ensuring inverse operations equate to transposition while maintaining vector lengths and angles. The orthogonal matrix consists of orthonormal column vectors $\{\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_n\}$, where each column satisfies the Kronecker symbol relationship:

$$\mathbf{q}_i \cdot \mathbf{q}_k = \delta_{ik},$$

equal to 1 when $i = k$ and 0 when $i \neq k$. Another important property is its ability to preserve angles and lengths through determinant, with $detQ = +1$ indicating pure rotation, and $detQ = -1$ indicating rotation combined with reflection.
One of the most geometrically significant properties of orthogonal matrices is their ability to preserve scalar products during transformations, as for any two vectors $\mathbf{x}$ and $\mathbf{y}$, the relationship $\mathbf{x} \cdot \mathbf{y} = Q\mathbf{x} \cdot Q\mathbf{y}$ holds. This preservation property guarantees that orthogonal matrices represent isometric transformations in $\mathbb{R}^n$, meaning they maintain the geometric integrity of objects during rotation or reflection operations. The transformation

$$T\colon \mathbb{R}^n \to \mathbb{R}^n$$
$$T(\mathbf{x}) = Q\mathbf{x}$$

is equivalent to a rotation of the coordinate system together with a possible reflection along some hyperplane.
To look at the object from many different angles, three matrices which represent rotations around each of the coordinate axes Ox, Oy, and Oz are used. Each rotation is represented by a matrix—R_x, R_y, R_z corresponding to the x-, y-, or z-axis. These individual rotation matrices can be multiplied to create composite transformations:

$$R = R_xR_yR_z.$$

Each elementary rotation matrix maintains orthogonality, and their product preserves this property, ensuring that complex 3D rotations maintain geometric property while enabling sophisticated object manipulation in three-dimensional space.
In computer graphics, most objects are not actually smooth but rather represented by polymesh. These polymesh are created by identifying vertices on the surface of an object

and connecting those vertices with edges, thus forming flat faces. Typically, these faces are triangular, as any three points in space uniquely define a plane, whereas four or more points generally do not. When the number of faces is large, the object, when viewed from a distance, appears smooth. Two primary components are required for creating three-dimensional computer models: the coordinates of the vertices, usually represented as an $n \times 3$ array V (where n is the number of vertices), and edges indicating connections between vertices. These edges can be represented by an $n \times n$ adjacency matrix E, or alternatively, by an $m \times 3$ array defining the vertices of each face, with m being the number of faces.
This project explores the theoretical foundations and practical implementations of orthogonal matrices for rotations in 3D graphics. The goal is to demonstrate how orthogonal matrices, through rotation matrices around the Ox, Oy, and Oz axes, can be utilized to visualize objects clearly and accurately from various perspectives. Such applications are crucial in realistic rendering in animations, interactive simulations, and scientific visualization tasks, ensuring numerical stability and computational efficiency in real-time environments.

2 Theoretical Foundation

This section explores the theoretical underpinnings of orthogonal matrices in linear algebra, focusing on their critical role in 3D graphics. Specifically, we examine how orthogonal matrices—due to their algebraic structure and geometric interpretation—enable precise and distortion-free transformations such as rotations, which are indispensable for realistic rendering and physical simulations.

2.1 Linear Transformations and Rotations

A mapping $T : \mathbb{R}^n \to \mathbb{R}^m$ is called a ***linear transformation*** if it satisfies two properties for all vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ and any scalar $c \in \mathbb{R}$ [2]:

$$T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v}), \quad T(c\mathbf{u}) = cT(\mathbf{u}).$$

Rotation, as a specific type of linear transformation, maps vectors in Euclidean space while preserving their lengths and relative angles. For any vector $\mathbf{x}$, a rotation transformation can be written in matrix form as:

$$\text{Rot}(\mathbf{x}) = R\mathbf{x}, \tag{1}$$

where R is the rotation matrix corresponding to the desired axis and angle of rotation.

2.2 Orthogonal Matrices

A matrix $Q \in \mathbb{R}^{n \times n}$ is ***orthogonal*** if its transpose equals its inverse:

$$Q^{-1} = Q^T$$

or, equivalent to

$$Q^T Q = QQ^T = I,$$

where I is the identity matrix. This definition implies that the rows and columns of Q form an orthonormal basis—i.e., they are all unit vectors and mutually perpendicular. A critical property of orthogonal matrices is that they preserve the Euclidean dot product [3]. For any vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$,

$$\mathbf{x} \cdot \mathbf{y} = (Q\mathbf{x}) \cdot (Q\mathbf{y}),$$

which in turn means that lengths and angles are preserved under orthogonal transformations. Consequently, orthogonal matrices implement *rigid-body transformations*, making them ideal for 3D graphics where maintaining shape and scale fidelity is essential.
The determinant of an orthogonal matrix is either $+1$ or -1. A determinant of $+1$ corresponds to a proper rotation, while -1 indicates a reflection combined with rotation. Most 3D rendering systems require proper rotations to ensure correct face orientation and consistent lighting behavior.

2.3 Rotation Matrices in 2D

Consider a vector $\mathbf{v} = (x, y)$ rotated counterclockwise by an angle θ around the origin. The new coordinates (x', y') are given by [3]:

$$x' = x\cos\theta - y\sin\theta, \quad y' = x\sin\theta + y\cos\theta.$$

In matrix form, this becomes:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

The direction of vector rotation is counterclockwise if θ is positive, and clockwise if θ is negative. Thus, the clockwise roation is found by replacing θ by $-\theta$ and using the trigonometric symmetric, we have the new coordinates for clockwise rotation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) \\ \sin(-\theta) & \cos(-\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

2.4 Rotation Matrices in 3D

In 3D space, rotations are typically decomposed into successive rotations around the principal axes x-, y-, and z-. The corresponding rotation matrices are: [3]
Rotation about the x-axis:

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

Rotation about the y-axis:

$$R_y(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix}.$$

Rotation about the z-axis:

$$R_z(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

By multiplying these in a specific order (e.g., $R = R_x R_y R_z$), we obtain a composite 3×3 orthogonal matrix that defines an arbitrary rotation in 3D space. The order of multiplication is crucial due to the non-commutative nature of matrix operations; altering the order yields different results.

Equation (1) expresses the rotation of a vector $\mathbf{x}$ in column-vector form, where the rotation is applied as $\text{Rot}(\mathbf{x}) = R\mathbf{x}$. However, in certain contexts—especially in computer graphics or numerical computation—vectors are often stored or processed as row vectors. In such cases, the corresponding transformation must be adapted accordingly.

Let $\mathbf{u}$ be a row vector in $\mathbb{R}^3$, and let $R = R_x R_y R_z$ be the composite rotation matrix formed by successive rotations about the x-, y-, and z-axes. To rotate $\mathbf{u}$ using matrix multiplication while keeping it as a row vector, we firstly take the transpose of both sides of Equation (1). If we consider the column-vector form $\mathbf{x}_{\text{rot}} = R\mathbf{x}$, then transposing both sides yields:

$$\mathbf{x}_{\text{rot}}^T = (R\mathbf{x})^T = \mathbf{x}^T R^T.$$

Thus, by identifying $\mathbf{u} = \mathbf{x}^T$ and $\mathbf{u}_{\text{rot}} = \mathbf{x}_{\text{rot}}^T$, we obtain the corresponding row-vector transformation as:

$$\mathbf{u}_{\text{rot}} = \mathbf{u}R^T. \tag{2}$$

Since R is an orthogonal matrix, we have $R^T = R^{-1}$, and hence the transformation $\mathbf{u}R^T$ preserves both lengths and angles, just like $R\mathbf{x}$ in the column-vector setting. This equivalence allows flexibility in implementation without compromising the geometric integrity of the rotation operation.

Example. Consider a unit cube centered at the origin, whose 8 vertices are represented as rows in a matrix $A \in \mathbb{R}^{8\times 3}$. Each row corresponds to the (x, y, z) coordinates of a vertex:

$$A = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

We would like to apply a composite 3D rotation to this cube using Euler angles $\theta_x = \frac{\pi}{3}$, $\theta_y = \frac{\pi}{4}$, and $\theta_z = \frac{\pi}{6}$ about the x-, y-, and z-axes respectively. The corresponding rotation matrices are:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\frac{\pi}{3} & -\sin\frac{\pi}{3} \\ 0 & \sin\frac{\pi}{3} & \cos\frac{\pi}{3} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ 0 & \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix},$$

$$R_y = \begin{bmatrix} \cos\frac{\pi}{4} & 0 & \sin\frac{\pi}{4} \\ 0 & 1 & 0 \\ -\sin\frac{\pi}{4} & 0 & \cos\frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix},$$

$$R_z = \begin{bmatrix} \cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 \\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The composite rotation matrix R is computed by matrix multiplication in the order:

$$R = R_x R_y R_z = \frac{1}{8}\begin{bmatrix} 2\sqrt{6} & -2\sqrt{2} & 4\sqrt{2} \\ 2+3\sqrt{2} & 2\sqrt{3}-\sqrt{6} & -2\sqrt{6} \\ 2\sqrt{3}-\sqrt{6} & 6+\sqrt{2} & 2\sqrt{2} \end{bmatrix}. \tag{3}$$

To obtain the rotated coordinates A_{rot}, we apply the equation (2). The resulting matrix A_{rot}, which contains the coordinates of the rotated cube in 3D space, is:

$$\begin{aligned} A_{\text{rot}} &= AR^T = \frac{1}{8}\begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & -1 \\ -1 & -1 & -1 \end{bmatrix} \begin{bmatrix} 2\sqrt{6} & 2+3\sqrt{2} & 2\sqrt{3}-\sqrt{6} \\ -2\sqrt{2} & 2\sqrt{3}-\sqrt{6} & 6+\sqrt{2} \\ 4\sqrt{2} & -2\sqrt{6} & 2\sqrt{2} \end{bmatrix} \\ &= \frac{1}{8}\begin{bmatrix} 2\sqrt{2}+2\sqrt{6} & 2+3\sqrt{2}+2\sqrt{3}-3\sqrt{6} & 6+3\sqrt{2}+2\sqrt{3}-\sqrt{6} \\ -2\sqrt{6}+4\sqrt{2} & -2-3\sqrt{2}+2\sqrt{3}-3\sqrt{6} & 6+3\sqrt{2}-2\sqrt{3}+\sqrt{6} \\ 6\sqrt{2}+2\sqrt{6} & 2+3\sqrt{2}-2\sqrt{3}-\sqrt{6} & -6+\sqrt{2}+2\sqrt{3}-\sqrt{6} \\ 2\sqrt{6}-6\sqrt{2} & 2+3\sqrt{2}+2\sqrt{3}+\sqrt{6} & 6-\sqrt{2}+2\sqrt{3}-\sqrt{6} \\ -2\sqrt{6}+6\sqrt{2} & -2-3\sqrt{2}-2\sqrt{3}-\sqrt{6} & -6+\sqrt{2}-2\sqrt{3}+\sqrt{6} \\ -6\sqrt{2}-2\sqrt{6} & -2-3\sqrt{2}+2\sqrt{3}+\sqrt{6} & 6-\sqrt{2}-2\sqrt{3}+\sqrt{6} \\ 2\sqrt{6}-2\sqrt{2} & 2+3\sqrt{2}-2\sqrt{3}+3\sqrt{6} & -6-3\sqrt{2}+2\sqrt{3}-\sqrt{6} \\ -2\sqrt{2}-2\sqrt{6} & -2-3\sqrt{2}-2\sqrt{3}+3\sqrt{6} & -6-3\sqrt{2}-2\sqrt{3}+\sqrt{6} \end{bmatrix}. \end{aligned} \tag{4}$$

3 Code

3.1 Cube problem

Task 1–4. Implementing a composite 3D rotation function.

In Tasks 1 through 4, we are required to implement a user-defined function with the header of `rotation(theta_x, theta_y, theta_z)` in `rotation.py`.
The motivation for constructing this function is to enable a flexible and mathematically precise way to apply sequential rotations around the x-, y-, and z-axes to any 3D object, such as the cube model defined earlier. Rather than applying each axis transformation manually or separately, we encapsulate this functionality within a reusable function that outputs a single 3×3 rotation matrix based on three input angles θ_x, θ_y, and θ_z. Listing 1 shows how we perform the computation.

```
import numpy as np

def rotation(theta_x, theta_y, theta_z):
    Rx = np.array([
        [1, 0, 0],
        [0, np.cos(theta_x), -np.sin(theta_x)],
        [0, np.sin(theta_x), np.cos(theta_x)]
    ])
    Ry = np.array([
```

```
        [np.cos(theta_y), 0, np.sin(theta_y)],
        [0, 1, 0],
        [-np.sin(theta_y), 0, np.cos(theta_y)]
    ])
    Rz = np.array([
        [np.cos(theta_z), -np.sin(theta_z), 0],
        [np.sin(theta_z), np.cos(theta_z), 0],
        [0, 0, 1]
    ])
    rotmat = Rx @ Ry @ Rz

    return rotmat
```

Listing 1: *Composite 3D rotation matrix*

Within the function, we define three standard rotation matrices: `Rx`, `Ry`, and `Rz` which represent counter-clockwise rotations around the x-, y-, and z-axes respectively. Each of these matrices is constructed using trigonometric identities involving sine and cosine, consistent with the definitions from linear algebra, which are followed by equation (2). These matrices are then composed using matrix multiplication in the order $R_x R_y R_z$ which effectively applies the z-axis rotation first, followed by the y-axis, and finally the x-axis. This specific multiplication sequence is intentional and determines how the object is ultimately rotated in 3D space. The order of multiplication matters because matrix multiplication is not commutative, and different orders produce different orientations. In Python, we use the `@` operator for those matrix multiplication, so the variable `rotmat` stores the final 3×3 rotation matrix. The meaning of `rotmat` is that it provides a coordinate transformation: when we multiply `rotmat` with a vector representing a 3D point, it gives us the coordinates of that point after being rotated around the three axes by the specified angles.

Task 5. Constructing the original cube.

We define a cube centered at the origin using eight vertices in three-dimensional space. Each vertex represents a corner of the cube, and the coordinates are chosen such that the cube spans from -1 to $+1$ along each axis. Therefore, the array `Vertices` will consist 8 sets of $(\pm 1, \pm 1, \pm 1)$. To model the structure of the cube, an 8×8 adjacency matrix is used to indicate the connectivity between vertices, where a value of 1 denotes an edge, they are stored in `Edges` variable. A zero matrix is first created, after which entries are set to 1 wherever two vertices share an edge. At this stage the adjacency is directed, because only the forward half of each vertex pair is filled; for example, the assignment `Edges[0, 1]` $= 1$ marks the edge $V_0 \to V_1$ but leaves $V_1 \to V_0$ unset. To convert the representation to an undirected graph, we add the matrix to its own transpose at the end (see List. 2).

```
import numpy as np

Vertices = np.array([
    [1, 1, 1], [-1, 1, 1], [1, -1, 1], [1, 1, -1],
    [-1, -1, 1], [-1, 1, -1], [1, -1, -1], [-1, -1, -1]
])

Edges = np.zeros((8, 8))
Edges[0, 1], Edges[0, 2], Edges[0, 3] = 1, 1, 1
```

```
Edges[1, 4], Edges[1, 5] = 1, 1
Edges[2, 4], Edges[2, 6] = 1, 1
Edges[3, 5], Edges[3, 6] = 1, 1
Edges[4, 7], Edges[5, 7], Edges[6, 7] = 1, 1, 1
Edges += Edges.T
```

Listing 2: *Definition of cube vertices and edges*

Task 6–7. Applying and visualizing 3D rotation.

Before resolving these tasks, we implement a helper function to to traverses the symmetric adjacency matrix `Edges` and draws line segments between connected vertices, named `plot_cube`. The code is shown in List. 3:

```
import matplotlib.pyplot as plt

def plot_cube(vertices, edges, title="Cube", figsize=(6, 6)):
    """Plots a 3D cube or its 2D projection."""
    fig = plt.figure(figsize=figsize)
    if vertices.shape[1] == 2:
        ax = fig.add_subplot(111)
        for i in range(8):
            for j in range(i + 1, 8):
                if edges[i, j] == 1:
                    ax.plot([vertices[i, 0], vertices[j, 0]],
                            [vertices[i, 1], vertices[j, 1]], 'b')
        ax.set_title(title)
        ax.set_aspect('auto')
        ax.set_xlabel('X')
        ax.set_ylabel('Z')
    else:
        ax = fig.add_subplot(111, projection='3d')
        for i in range(8):
            for j in range(i + 1, 8):
                if edges[i, j] == 1:
                    ax.plot([vertices[i, 0], vertices[j, 0]],
                            [vertices[i, 1], vertices[j, 1]],
                            [vertices[i, 2], vertices[j, 2]], 'b')
        ax.set_title(title)
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        ax.set_box_aspect([1, 1, 1])

    plt.show()
```

Listing 3: *Visualization function for plotting the cube structure in 2D or 3D*

This algorithm works by first checking whether the input `vertices` array represents 2D or 3D data, based on the number of columns. If the array has two columns, the cube is projected onto a 2D plane and rendered using a standard 2D plot. Otherwise, the cube is plotted in full 3D using the appropriate 3D plotting backend of Matplotlib. The function iterates over all vertex pairs (i, j) with $i < j$ and draws a line between them whenever the adjacency matrix entry `edges[i, j]` equals 1. This ensures that each edge of the cube is drawn exactly once.

See List. 4, in this stage the objective is to demonstrate that the user-defined function rotation can be used to re-orient a solid object in three-dimensional space and to verify the transformation visually. We begin by selecting three non-trivial angles, $\theta_x = \pi/3$, $\theta_y = \pi/4$, and $\theta_z = \pi/6$, which guarantee that the cube is rotated out of alignment with every coordinate plane and thus provide an unambiguous test of the composite rotation matrix. These angles are passed to `rotation` function, yielding the 3×3 matrix `rotmat`. Because each vertex of the cube is stored as a row vector in the array `Vertices`, the matrix is applied on the right and transposed `rotmat.T` so that the standard algebraic convention of post-multiplying row vectors is respected. The transformed vertex array `VertRot = Vertices @ rotmat.T` contains the coordinates of the cube after a sequential rotation. Both the original and rotated data sets are then forwarded to `plot_cube` function as mentioned above.

```
theta1, theta2, theta3 = np.pi / 3, np.pi / 4, np.pi / 6
rotmat = rotation(theta1, theta2, theta3)
VertRot = Vertices @ rotmat.T
print(f"Rotation matrix:\n{rotmat}")
print(f"Rotated vertices:\n{VertRot}")

plot_cube(Vertices, Edges, title="Original Cube")
plot_cube(VertRot, Edges, title="Rotated Cube")
```

Listing 4: *Applying composite rotation to 3D cube and plotting results*

The printed rotation matrix and rotated vertices matrix are expressed in List 5:

```
Rotation matrix:
[[ 0.61237244 -0.35355339  0.70710678]
 [ 0.78033009  0.12682648 -0.61237244]
 [ 0.12682648  0.9267767   0.35355339]]
Rotated vertices:
[[ 0.96592583  0.29478413  1.40715657]
 [-0.25881905 -1.26587604  1.1535036 ]
 [ 1.67303261  0.04113117 -0.44639682]
 [-0.44828774  1.51952901  0.70004979]
 [ 0.44828774 -1.51952901 -0.70004979]
 [-1.67303261 -0.04113117  0.44639682]
 [ 0.25881905  1.26587604 -1.1535036 ]
 [-0.96592583 -0.29478413 -1.40715657]]
```

Listing 5: *Printed output of rotation matrix and rotated vertices*

This result is approximately equal to the numerical calculations in (3) and (4). It confirms that our `rotation` function works properly.
The plotting outputs are illustrated in Fig. 1 and Fig. 2, respectively. These visualizations confirm the effectiveness of the transformation: the “Original Cube” remains aligned with the coordinate axes, while the “Rotated Cube” appears in a skewed orientation, verifying that all three axis rotations were applied.

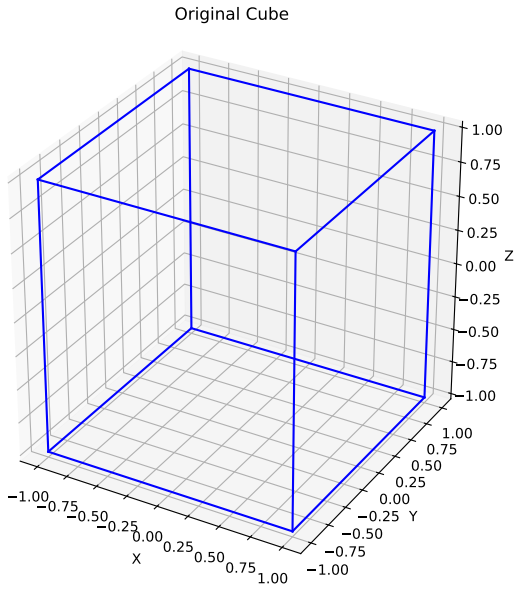

Figure 1: *Plotting result of original cube*

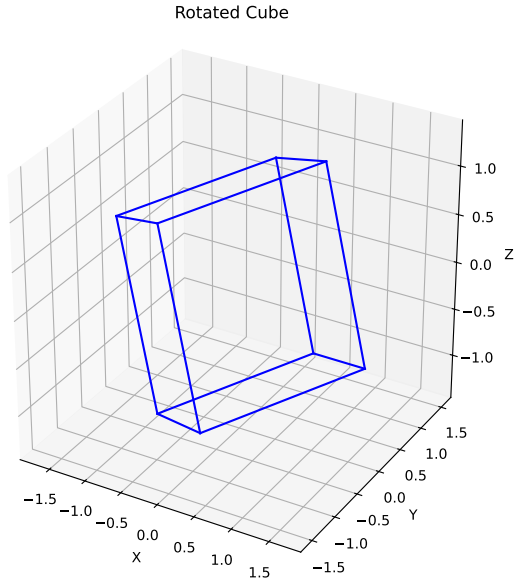

Figure 2: *Plotting result of rotated cube*

Task 8–9. Projecting rotated cube onto 2D plane.

In this stage, we aim to generate and visualize a two-dimensional projection of the already rotated cube by discarding one of its spatial coordinates.

Question 1: Why does the second `for` cycle start with $j + 1$?

We begin by addressing Question 1, which concerns the nested loops in the projection logic. The inner loop starts from index $j + 1$ rather than 0 to avoid redundant drawing. Since the adjacency matrix `Edges` is symmetric, each edge is represented twice (i.e., both `Edges[i, j]` and `Edges[j, i]` are 1 for a valid edge). Iterating from $j + 1$ ensures that each unique edge is processed exactly once and prevents duplicated lines in the output.

To implement the projection, we drop the second coordinate (the y-axis) from each vertex in `VertRot`, thereby flattening the cube into the Oxz plane. See List. 6, the dropping is accomplished in line 1. The resulting 2D array `VertRotPrj` is then passed to the previously defined `plot_cube` function, which automatically detects the 2D input and generates a plot accordingly.

```
1 VertRotPrj = VertRot[:, [0, 2]]
2
3 plot_cube(VertRotPrj, Edges, title="Projection of Rotated Cube")
```

Listing 6: *Projecting the rotated cube onto the Oxz plane*

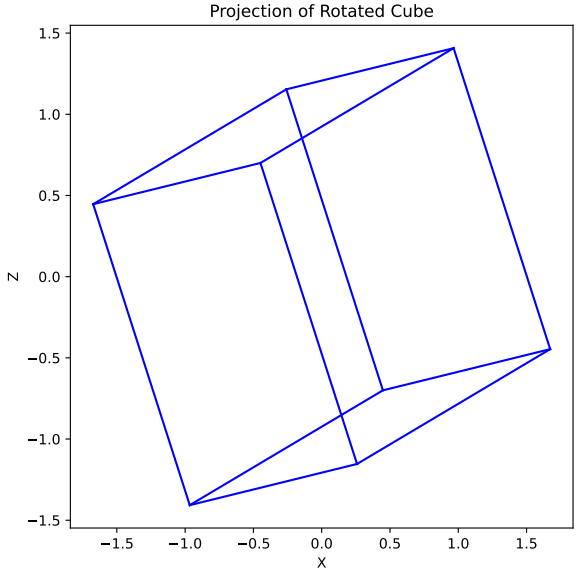

Figure 3: *2D projection of the rotated cube onto the Oxz plane*

The output, shown in Fig. 3, is a flat projection of the 3D cube onto the Oxz plane. This projection allows us to inspect the spatial structure of the rotated cube from a top-down view, making it easier to analyze geometric relationships and symmetry from that perspective.

3.2 Bucky Ball problem

Task 10–11. Generating and visualizing a Buckminsterfullerene.

Before moving to the main tasks' requirement, we build the `bucky()` procedure to illustrate a truncated icosahedron mathematically.

```
1 import numpy as np
2
3 def bucky():
4     # Start with the vertices of a regular icosahedron
5     phi = (1 + np.sqrt(5)) / 2  # Golden ratio
6
7     # The 12 vertices of a regular icosahedron
8     icosa_vertices = []
9     for i in [-1, 1]:
```

```
        icosa_vertices.append([0, i, phi])
        icosa_vertices.append([0, i, -phi])
        icosa_vertices.append([phi, 0, i])
        icosa_vertices.append([-phi, 0, i])
        icosa_vertices.append([i, phi, 0])
        icosa_vertices.append([i, -phi, 0])

    icosa_vertices = np.array(icosa_vertices)

    # Edges of the icosahedron (30 edges)
    icosa_edges = []
    threshold = 2.1

    for i in range(len(icosa_vertices)):
        for j in range(i+1, len(icosa_vertices)):
            dist = np.linalg.norm(icosa_vertices[i] - icosa_vertices[j])
            if dist < threshold:
                icosa_edges.append((i, j))

    # For a truncated icosahedron, we keep 1/3 of each edge from each
    vertex
    trunc_factor = 1/3

    vertices = []

    # For each edge, compute two new vertices (one for each endpoint)
    for i, j in icosa_edges:
        v_i = icosa_vertices[i]
        v_j = icosa_vertices[j]

        t_i = v_i + trunc_factor * (v_j - v_i)
        vertices.append(t_i)

        t_j = v_j + trunc_factor * (v_i - v_j)
        vertices.append(t_j)

    vertices = np.array(vertices)

    # Normalize vertices to place them on a unit sphere
    norms = np.linalg.norm(vertices, axis=1)
    vertices = vertices / norms[:, np.newaxis]

    # Remove duplicate vertices (with small tolerance)
    unique_vertices = []
    tolerance = 1e-10

    for v in vertices:
        is_duplicate = False
        for u in unique_vertices:
            if np.linalg.norm(v - u) < tolerance:
                is_duplicate = True
                break
        if not is_duplicate:
            unique_vertices.append(v)

    vertices = np.array(unique_vertices)
```

```
65
66      # Generate edges by connecting vertices that are close to each other
67      edges = []
68      threshold = 0.6  # Adjusted for C60 structure
69
70      for i in range(len(vertices)):
71          for j in range(i+1, len(vertices)):
72              dist = np.linalg.norm(vertices[i] - vertices[j])
73              if dist < threshold:
74                  edges.append((i, j))
75
76      return vertices, edges
```

Listing 7: *Synthesises the 60-vertex truncated icosahedron in the* `bucky` *procedure*

The algorithm in List. 7 is inspired by the relationship between C_{60} structure and golden ratio $\phi = (1 + \sqrt{5})/2$ [4]. Since the icosahedron is the geometric base shape for the truncated icosahedron. Its vertices provide a framework from which we create the fullerene structure. Lines 4–17 define the golden ratio ϕ and construct the 12 vertices of a regular icosahedron, using permutations of ± 1 and $\pm\phi$ along the x-, y-, and z-axes. Lines 19–27 compute edges by checking all vertex pairs: if their Euclidean distance is less than 2.1, the pair is recorded as an edge (a tuple of vertex indices). Lines 29–49 generate two new vertices for each edge (v_i, v_j), located one-third along the edge from each endpoint. These truncation points are stored in a list, converted to an array, and normalized to lie on the unit sphere by dividing each by its norm. Lines 51–64 remove duplicates by checking whether a vertex lies within a tolerance of 10^{-10} from any in the unique list. This ensures numerical precision while preventing unintended merges. Lines 66-74 construct the adjacency of the truncated icosahedron by connecting pairs of unique vertices whose distance is below 0.6, consistent with the expected edge length of a buckyball.
Invoking this routine in the main script (see List. 8) fulfils the tasks' requirement. We retrieve the geometry, plot it with the previously defined `plot_buckyball` helper (we will explain this function in the next part), and verify the vertex count:

```
Vertices2, Edges2 = bucky()
plot_buckyball(Vertices2, Edges2)
Vertices2Num = Vertices2.shape[0]
print(f"Number of vertices in the buckyball: {Vertices2Num}")
Vert2Rot = Vertices2 @ rotmat.T
Vert2Prj = Vert2Rot[:, [0, 1]]
plot_buckyball(Vert2Prj, Edges2, title="2D Projection of Buckyball")
```

Listing 8: *Acquiring and viewing the buckyball*

The resulting figure (see Fig. 4) shows the familiar soccer-ball framework—a mesh of twelve pentagons and twenty hexagons whose 60 vertices are evenly distributed on a sphere. The console output "Number of vertices in the buckyball: 60" confirms that the algorithm reproduces the exact vertex count prescribed by the fullerene's combinatorial structure, thereby validating both the geometric construction and the edge-generation heuristics.

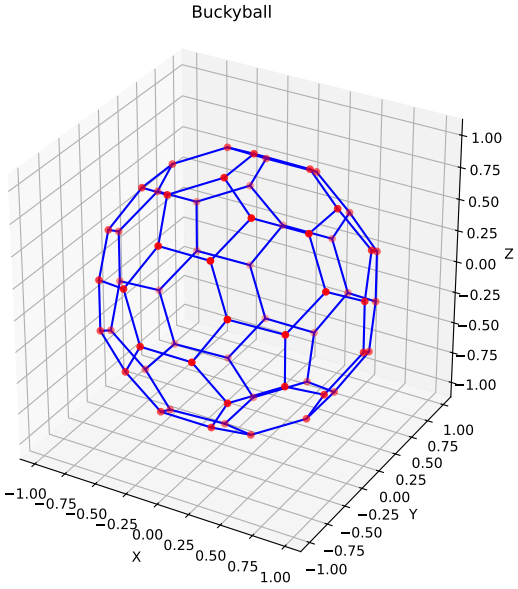

Figure 4: *Plotting result of a 3D bucky ball*

When projecting a 3D bucky ball onto 2D plane, we see a symmetrical network of pentagons and hexagons, similar to a soccer ball pattern, but projected flat onto the screen, as shown in Fig. 5.

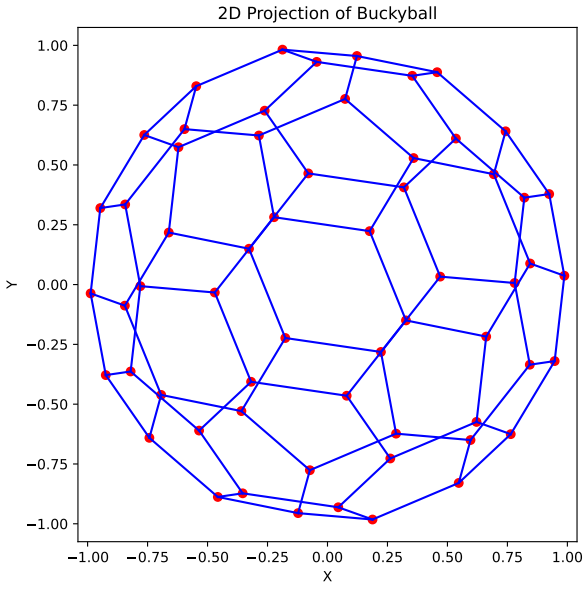

Figure 5: *2D projection of the rotated bucky ball onto the Oxy plane*

Task 12. Visualizing the 3D structure of the buckyball.

We return to the helper function `plot_buckyball`, as showned in List. 9.

```
import matplotlib.pyplot as plt

def plot_buckyball(vertices, edges, title="Buckyball", figsize=(6, 6)):
    """Plots the buckyball structure in 3D or its 2D projection."""
```

```
    fig = plt.figure(figsize=figsize)

    if vertices.shape[1] == 3:
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(vertices[:, 0], vertices[:, 1], vertices[:, 2], c='r'
    , marker='o')

        for edge in edges:
            x_vals = [vertices[edge[0]][0], vertices[edge[1]][0]]
            y_vals = [vertices[edge[0]][1], vertices[edge[1]][1]]
            z_vals = [vertices[edge[0]][2], vertices[edge[1]][2]]
            ax.plot(x_vals, y_vals, z_vals, c='b')

        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_zlabel('Z')
        ax.set_title(title)
        ax.set_box_aspect([1, 1, 1])
    else:
        ax = fig.add_subplot(111)
        ax.scatter(vertices[:, 0], vertices[:, 1], c='r', marker='o')

        for edge in edges:
            x_vals = [vertices[edge[0]][0], vertices[edge[1]][0]]
            y_vals = [vertices[edge[0]][1], vertices[edge[1]][1]]
            ax.plot(x_vals, y_vals, c='b')

        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_title(title)
        plt.axis('equal')

    plt.show()
```

Listing 9: *Visualization function for plotting the buckyball structure in 2D or 3D*

Question 2: Why do we use 60 as the end limit for our `for` cycles?
In the truncated-icosahedron model every vertex represents one carbon atom of the C_{60} molecule, so the array `Vertices2` contains exactly 60 rows—one 3-tuple of Cartesian coordinates for each atom. For that reason the MATLAB loops in Task 12 run from $j = 1$ to 60, and $k = j + 1$ to 60: the upper bound guarantees that every vertex is visited once, and no attempt is made to access an index that lies outside the array. Any smaller limit would omit vertices (and therefore edges); any larger limit would raise an out-of-bounds error. Thus the value 60 is dictated directly by the combinatorial structure of the buckyball itself.

3.3 Low Poly Fawn problem

Task 13–14. Reading 3D Fawn model data.

In these tasks, the purpose of the script is to read the structure of a three-dimensional model prepared for 3D printing by examining its face matrix `f`, which encodes the surface geometry of the object. The matrices `v` and `f` are loaded from MATLAB .mat files, where `v` contains the spatial coordinates of the model's vertices, and `f` specifies how

these vertices are grouped to form the triangular faces of the mesh. Since MATLAB uses one-based indexing and Python uses zero-based indexing, the face matrix `f` must be adjusted by subtracting one to ensure that vertex indices correctly reference entries in the Python-based vertex array (see List. 10).

```
import scipy.io

v_mat = scipy.io.loadmat('v.mat')
f_mat = scipy.io.loadmat('f.mat')
v = v_mat['v']
f = f_mat['f'] - 1

mFaces, nFaces = f.shape
print(f"Number of faces (mFaces): {mFaces}, Number of vertices per face (nFaces): {nFaces}")
```

Listing 10: *Reading 3D Fawn model data*

Once the data is loaded and corrected, the shape of the face matrix is examined to determine two key structural parameters: `mFaces`, representing the total number of triangular faces in the model, and `nFaces`, representing the number of vertices per face. The resulting output "Number of faces (mFaces): 440, Number of vertices per face (nFaces): 3" confirms that the mesh is composed entirely of triangles, a standard representation in 3D computer graphics and additive manufacturing.

Task 15–17. Visualization and projection of a 3D fawn model.

Before resolving the main tasks, we have two helper functions, which are `plot_3d_model` and `plot_2d_projection`, serve to visualize a 3D geometric model and its 2D projection, respectively. Both rely on the face-vertex representation of a 3D mesh, where each triangular face is defined by three indices pointing to rows in a vertex array. The former function is shown in List. 11.

```
import matplotlib.pyplot as plt

def plot_3d_model(vertices, faces, title="3D Model with Edges", figsize=(6, 6)):
    """Plots a 3D model with edges using faces and vertices."""
    fig = plt.figure(figsize=figsize)
    ax = fig.add_subplot(111, projection='3d')

    for j in range(faces.shape[0]):
        for k in range(3):
            x_vals = [vertices[faces[j, k], 0], vertices[faces[j, (k + 1) % 3], 0]]
            y_vals = [vertices[faces[j, k], 1], vertices[faces[j, (k + 1) % 3], 1]]
            z_vals = [vertices[faces[j, k], 2], vertices[faces[j, (k + 1) % 3], 2]]
            ax.plot(x_vals, y_vals, z_vals, 'b')

    ax.set_title(title)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
```

```
    ax.set_box_aspect([1, 1, 1])

    plt.show()
```

Listing 11: *Function to render a 3D mesh wireframe*

The `plot_3d_model` function constructs a 3D rendering of the model by iterating through each triangular face in the faces array. For each face, it plots the three edges of the triangle in 3D space using the coordinates of the corresponding vertices. The modulo operation $(k + 1)\%3$ ensures that edges wrap around from the third vertex back to the first, completing the triangle. On the other hand, the latter function is shown in List. 12.

```
import matplotlib.pyplot as plt

def plot_2d_projection(vertices, faces, title="2D Projection of 3D Model", figsize=(6, 6)):
    """Plots a 2D projection of a 3D model."""
    fig, ax = plt.subplots(figsize=figsize)

    for j in range(faces.shape[0]):
        for k in range(3):
            ax.plot([vertices[faces[j, k], 0], vertices[faces[j, (k + 1) % 3], 0]],
                    [vertices[faces[j, k], 1], vertices[faces[j, (k + 1) % 3], 1]], 'b')

    ax.set_title(title)
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    plt.axis('equal')
    plt.show()
```

Listing 12: *Function to plot a 2D projection of a 3D mesh*

The `plot_2d_projection` function performs a similar process but within 2D space. It assumes the vertex input has already been rotated and projected onto the Oxy plane. For each face, the function draws edges between consecutive vertices, just like in the 3D version, but using only 2D plotting.
Returning to the main tasks, the Python script to solve them is as below:

```
plot_3d_model(v, f, title="3D Model of Fawn")
vRot = v @ rotmat.T
vRotPrj = vRot[:, [0, 1]]
plot_2d_projection(vRotPrj, f, title="2D Projection of 3D Fawn")

# Alternative 2D projection using a different rotation matrix
rotmat2 = rotation(-np.pi / 3, 0, np.pi / 4)
vRot = v @ rotmat2.T
vPrj = vRot[:, [0, 1]]
plot_2d_projection(vPrj, f, title="Alternative 2D Projection of 3D Fawn")
```

Listing 13: *3D rendering and 2D projections using orthogonal rotations of the fawn model*

In List. 13, from line 1 to line 4, the code first generates a 3D visualization of the fawn model by plotting the edges of its triangular faces using the original vertex coordinates. This corresponds to the task of rendering the 3D mesh structure (see Fig. 6).

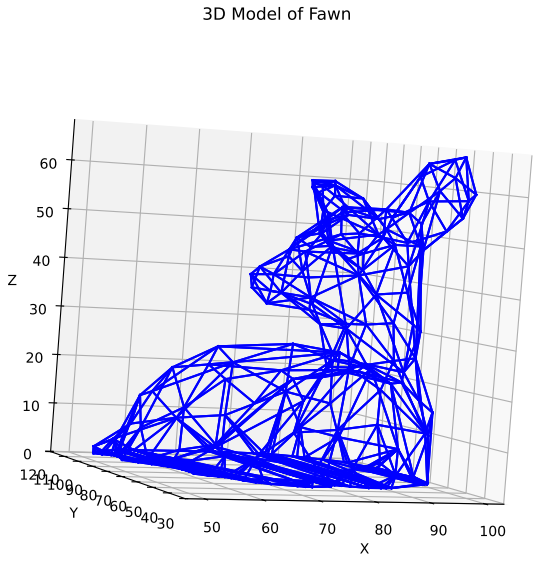

Figure 6: *3D wireframe model of the fawn*

Then, the vertices are rotated by a predefined rotation matrix `rotmat` and projected onto the Oxy plane by selecting the first two coordinate components. This transformed vertex set is used to produce a 2D projection of the rotated 3D model, enabling visualization from a specified viewpoint (see Fig. 7). The resulting plot illustrates how the spatial geometry of the model appears when flattened onto a plane after rotation.

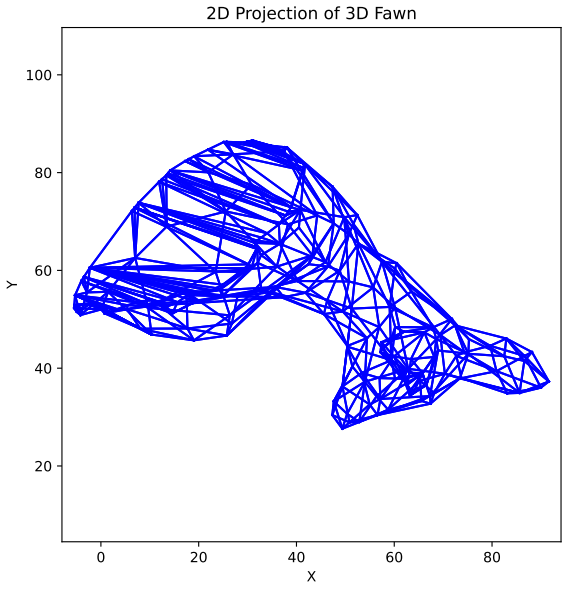

Figure 7: *2D projection of 3D Fawn after initial rotation*

From line 6 to the end, an alternative 2D projection is generated by applying a different rotation matrix `rotmat2`, which corresponds to a distinct orientation of the model in space. The vertices are rotated accordingly and projected onto the Oxy plane in the same manner as before. This produces a second 2D visualization that offers a new perspective on the model's geometry (see Fig. 8).

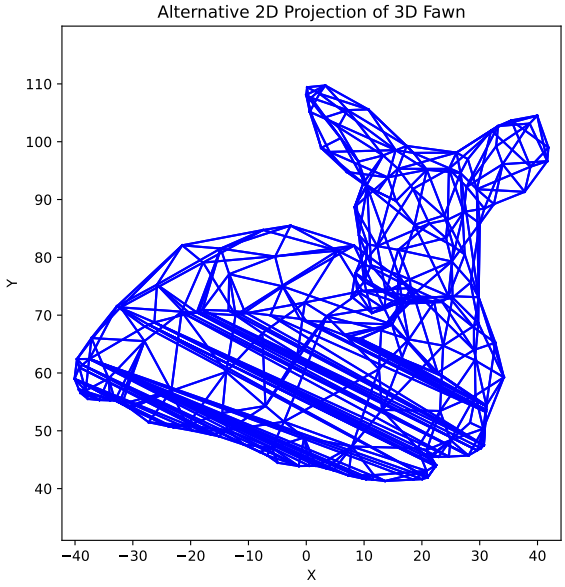

Figure 8: *2D projection of 3D Fawn after alternative rotation*

4 Illustrative Example

We have already verified the output of our code using the example provided in the assignment specification, confirming that the implementation produces correct and expected results. Therefore, we state that the code is functioning properly.
Now, we build a convex hull from randomly generated 3D points. Using NumPy, we first sample 30 points uniformly in the cube $[-5, 5]^3$ then apply the `scipy.spatial.ConvexHull` function to compute the convex hull enclosing these points. The hull is represented by its vertices and triangular faces, obtained from the `simplices` attribute. These data are passed to our plotting functions to visualize the original shape, its rotated version, and its 2D projection (see List. 14).

```
import numpy as np
from scipy.spatial import ConvexHull

np.random.seed(42)
points = np.random.rand(30, 3) * 10 - 5  # Random points in a cube [-5, 5]^3

hull = ConvexHull(points)
vertices = points
faces = hull.simplices

plot_3d_model(vertices, faces, title="Original Convex Hull")
rotmat = rotation(np.pi/6, np.pi/4, np.pi/3)
rotated_vertices = vertices @ rotmat.T
plot_3d_model(rotated_vertices, faces, title="Rotated Convex Hull")

projected_vertices = rotated_vertices[:, [0, 1]]
plot_2d_projection(projected_vertices, faces, title="2D Projection of 3D Convex Hull")
```

Listing 14: *Generate a convex hull from random 3D points*

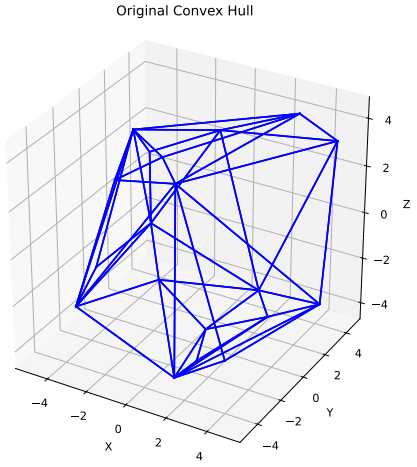

Figure 9: *Original convex hull generated from 30 random points in 3D space*

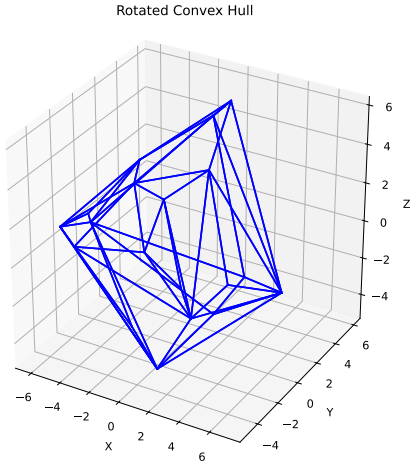

Figure 10: *Convex hull after applying an orthogonal rotation matrix*

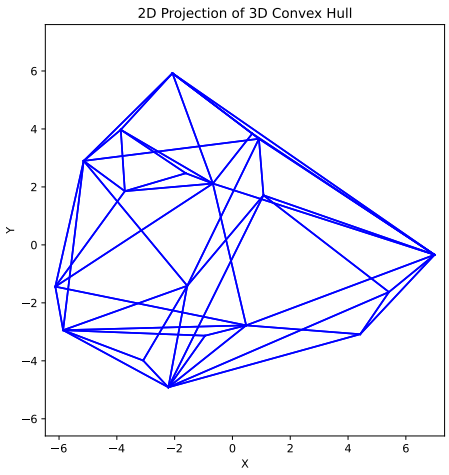

Figure 11: *2D projection of the rotated convex hull onto the Oxy plane*

The original, rotated and projection onto Oxy plane of a convex hull are illustrated in

Fig. 9, Fig. 10, and Fig. 11, respectively. They are showing that our code is running well. These visualizations demonstrate how an orthogonal transformation preserves the geometric structure of the 3D shape while enabling it to be viewed from a different angle. The 2D projection captures the outline of the shape as seen from the top-down perspective on the Oxy plane.

5 Conclusion and Further Discussion

5.1 Conclusion

In this assignment, we set out to transform sets of three-dimensional vertices into coherent two-dimensional projections. Our primary tool was a unified rotation function that applied sequential rotations around the x-, y-, and z-axes. Although this process initially felt unfamiliar, we quickly observed that the objects' proportions remained intact throughout—none of the cube, buckyball, or "fawn" mesh appeared distorted or skewed. In other words, the use of orthogonal matrices, which satisfy the condition $Q^TQ = I$, effectively preserved shape and size during all transformations.

One of the most insightful aspects of the project was realizing how a few matrix multiplications could generate accurate and meaningful wireframes. Initially, we were uncertain about how to correctly define adjacency between vertices. However, once we explicitly listed the vertex connections, plotting the corresponding edges became a straightforward task. Watching the buckyball rotate without any deformation reinforced the fundamental principle of rigid body transformations: all our rotations were orthogonal, and thus all were length- and angle-preserving.

We also learned the practical rule-of-thumb for rendering 2D views: first, apply rotation to all vertices; then, project by omitting the axis perpendicular to the desired view (e.g., z for the front view, x for the side view, etc.). This approach worked consistently across all examples, from the simple cube to the 60-vertex buckyball and the more complex "fawn" mesh. The fact that no extra tweaks were required emphasized the robustness and generality of this technique. Overall, this project demonstrated that even those new to 3D graphics can effectively use linear algebra to produce meaningful visualizations. By consistently applying orthogonal rotations followed by coordinate projection, we created clear and accurate front, side, and top views of each model. This process not only solidified our understanding of theoretical concepts such as orthogonal transformations, but also gave us hands-on experience and confidence in applying them—laying a strong foundation for future explorations in computer graphics and related fields.

5.2 Further Discussion

Several areas remain unexplored. For instance, we did not implement camera-based perspective projection, lighting models, shading, or hidden surface removal—all of which are essential for realistic 3D rendering. Additionally, while our rotation logic works well for static transformations, we did not extend it to support animation or interactive manipulation (e.g., using mouse input or GUI controls).

Future development could take several directions. A natural extension would be to incorporate perspective projection, where depth perception is more realistic than in orthographic views. Another useful direction would be to add interactive visualization-allowing

users to rotate, zoom, or switch projections dynamically. From a computational standpoint, integrating quaternion-based rotation could offer numerical stability and efficiency, particularly for complex animations. Finally, bridging this rotation-projection pipeline with real-time rendering frameworks would enable the transition from offline analysis to interactive 3D applications, laying groundwork for more advanced exploration in computer graphics or virtual/augmented reality systems.

Code Availability

The complete implementation of this assignment tasks, including all source code and related files, is available on GitHub. Please access the repository via https://github.com/longlephamtien/LA242-Assignment.

Acknowledgement

We would like to express our sincere gratitude to PhD. Dau The Phiet for valuable guidance and support throughout this study.

References

[1] P. Comnios, "Viewing and Projection Transformations," in *Mathematical and Computer Programming Techniques for Computer Graphics*. London: Springer, 2006, pp. 253–289.

[2] H. Anton and C. Rorres, "General Linear Transformations," in *Elementary Linear Algebra: Applications Version*. USA: Wiley, 2014, pp. 447–489.

[3] H. Anton and C. Rorres, "Diagonalization and Quadratic Forms," in *Elementary Linear Algebra: Applications Version*. USA: Wiley, 2014, pp. 401–445.

[4] G. Meisner, *Bucky Balls and Phi*, Accessed: 2025-05-02, May 2012. [Online]. Available: https://www.goldennumber.net/bucky-balls/.

Appendix: Mathematical Proofs

1 Proof that Rotation is a Linear Transformation

A transformation $T : \mathbb{R}^n \to \mathbb{R}^n$ is linear if:

$$T(\mathbf{u} + \mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v}),$$
$$T(c\mathbf{u}) = cT(\mathbf{u}).$$

Let $T(\mathbf{x}) = R\mathbf{x}$ for some fixed matrix $R \in \mathbb{R}^{n\times n}$.

$$T(\mathbf{u} + \mathbf{v}) = R(\mathbf{u} + \mathbf{v}) = R\mathbf{u} + R\mathbf{v} = T(\mathbf{u}) + T(\mathbf{v}),$$
$$T(c\mathbf{u}) = R(c\mathbf{u}) = c(R\mathbf{u}) = cT(\mathbf{u}).$$

Thus, T is a linear transformation.

2 Proof that Orthogonal Matrices Preserve Dot Products

Let $Q \in \mathbb{R}^{n\times n}$ be orthogonal: $Q^TQ = I$. For any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$,

$$\begin{aligned}(Q\mathbf{x}) \cdot (Q\mathbf{y}) &= (Q\mathbf{x})^T(Q\mathbf{y}) \\ &= \mathbf{x}^T Q^T Q\mathbf{y} \\ &= \mathbf{x}^T I\mathbf{y} = \mathbf{x} \cdot \mathbf{y}.\end{aligned}$$

Therefore, orthogonal transformations preserve inner products, and hence lengths and angles.

3 Derivation of 2D Rotation Matrix

Let $\mathbf{v} = (x, y)$ be a vector. Represent $\mathbf{v}$ in polar coordinates as:

$$x = r\cos\phi, \quad y = r\sin\phi.$$

After a counterclockwise rotation by angle θ, the new angle is $\phi + \theta$:

$$\begin{aligned}x' &= r\cos(\phi+\theta) = r(\cos\phi\cos\theta - \sin\phi\sin\theta), \\ y' &= r\sin(\phi+\theta) = r(\sin\phi\cos\theta + \cos\phi\sin\theta).\end{aligned}$$

Using $x = r\cos\phi$, $y = r\sin\phi$, we substitute:

$$\begin{aligned}x' &= x\cos\theta - y\sin\theta, \\ y' &= x\sin\theta + y\cos\theta.\end{aligned}$$

In matrix form:

$$\begin{bmatrix}x' \\ y'\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}\begin{bmatrix}x \\ y\end{bmatrix}.$$

4 Derivation of Rotation Matrices Along Principal Axes

Rotations in 3D space can be expressed as linear transformations using 3×3 orthogonal matrices. These are derived by rotating a point around one of the principal coordinate axes—x-, y-, or z-axis—while keeping the other coordinates invariant or updated according to a 2D rotation in the relevant plane.

Rotation About the x-Axis

Let a point in 3D space be represented by the vector

$$\mathbf{v} = \begin{bmatrix}x \\ y \\ z\end{bmatrix}.$$

When rotating $\mathbf{v}$ counterclockwise by angle θ_x around the x-axis, the x-component remains unchanged, while the y and z coordinates transform as if in a 2D rotation in the yz-plane:

$$\begin{aligned}y' &= y\cos\theta_x - z\sin\theta_x, \\ z' &= y\sin\theta_x + z\cos\theta_x.\end{aligned}$$

Thus, the rotated vector becomes:

$$\mathbf{v}' = \begin{bmatrix} x \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y\cos\theta_x - z\sin\theta_x \\ y\sin\theta_x + z\cos\theta_x \end{bmatrix}.$$

In matrix form:

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

Rotation About the y-Axis

Similarly, rotating a vector around the y-axis keeps the y-component unchanged, while the x and z components undergo a 2D rotation in the xz-plane:

$$\begin{aligned} x' &= x\cos\theta_y + z\sin\theta_y, \\ z' &= -x\sin\theta_y + z\cos\theta_y. \end{aligned}$$

So the transformed vector is:

$$\mathbf{v}' = \begin{bmatrix} x' \\ y \\ z' \end{bmatrix} = \begin{bmatrix} x\cos\theta_y + z\sin\theta_y \\ y \\ -x\sin\theta_y + z\cos\theta_y \end{bmatrix}.$$

In matrix form:

$$R_y(\theta_y) = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix}.$$

Rotation About the z-Axis

For a rotation around the z-axis, the z-component remains unchanged, while the x and y coordinates rotate in the xy-plane:

$$\begin{aligned} x' &= x\cos\theta_z - y\sin\theta_z, \\ y' &= x\sin\theta_z + y\cos\theta_z. \end{aligned}$$

The resulting vector is:

$$\mathbf{v}' = \begin{bmatrix} x' \\ y' \\ z \end{bmatrix} = \begin{bmatrix} x\cos\theta_z - y\sin\theta_z \\ x\sin\theta_z + y\cos\theta_z \\ z \end{bmatrix}.$$

In matrix form:

$$R_z(\theta_z) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$