Supplemental Materials for "A Unified Discrete Collision Framework for Triangle Primitives"

Submission ID: paper1071

1 Detailed Examination of Cases Where the Projected Triangle Degenerates.

Examples are provided for cases where two triangles intersect perpendicularly and degenerate upon projection, categorized according to their corrected contact states: Point-Triangle, Edge-Triangle, Point-Point, Edge-Edge, and Point-Edge cases.

The figure shows the step-by-step process for each example where two triangles intersect perpendicularly. The normal vector of the red-bordered triangle is (0,1,0), and the normal vector of the blue-bordered triangle is (0,0,1). (1) Initial state. (2) Project the triangles along the normal vector of the second triangle (corresponding to Step 1 in Section 3.3 of the paper). (3) Compute candidate points, calculate signed distances at each point, and determine the penetration depth for the first triangle (corresponding to Steps 2, 3, and 7 in Section 3.3 of the paper). (4)-(5) Perform the same processing for the second triangle (corresponding to Steps 4-6 and 8 in Section 3.3 of the paper). (6) Use Equation (6) from the paper to calculate the penetration depths of both triangles, as obtained in Steps 7 and 8 in Section 3.3 of the paper. Select the candidate point of the second triangle with the smallest penetration depth magnitude. (7) Apply the gradient, which is the normal vector of the first triangle, to resolve the collision (corresponding to Step 9 in Section 3.3 of the paper). (8) Top view of the figure: blue indicates the state before resolution and yellow indicates the state after resolution. In this figure, a simple push-out correction is applied to one of the triangles. The meaning of the step numbers is consistent across all figures.

1.1 Point-Triangle case

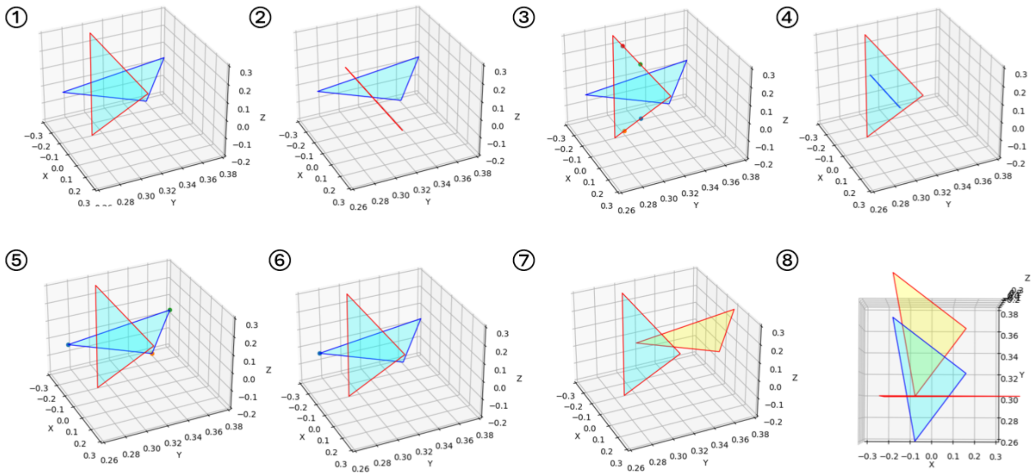

Figure 1: Point-Triangle case when the projected line segment fits inside the other triangle.

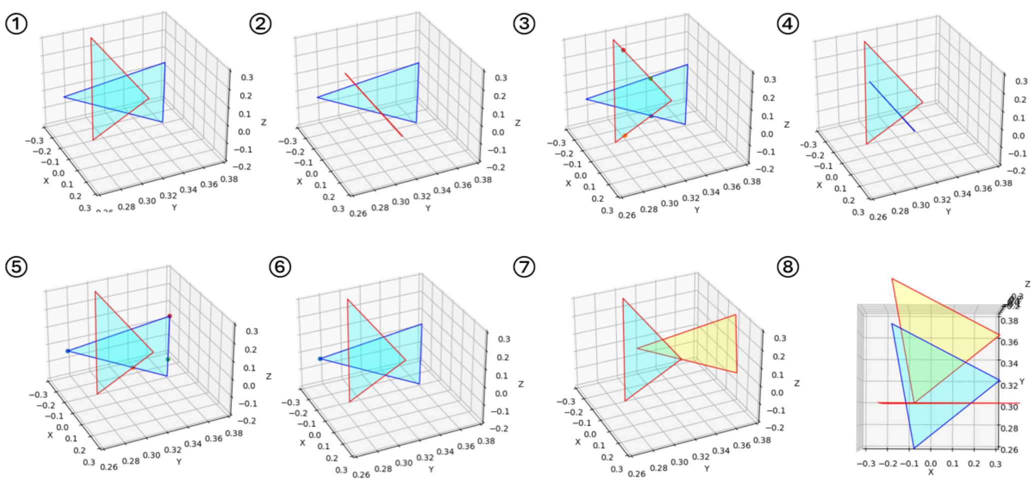

Figure 2: Point-Triangle case when the projected line segment does not fit inside the other triangle.

1.2 Edge-Triangle case

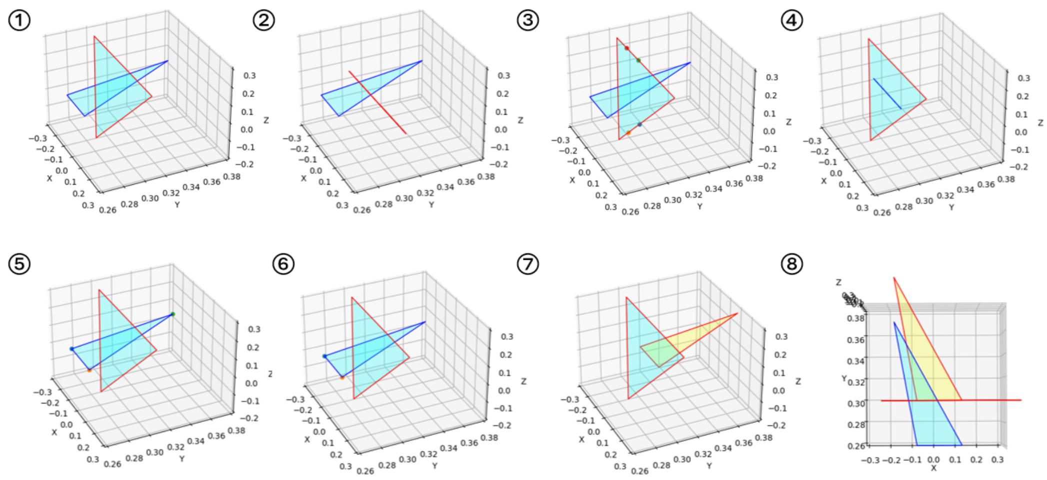

Figure 3: Edge-Triangle case when the projected line segment fits inside the other triangle. (The case is also included in the paper.)

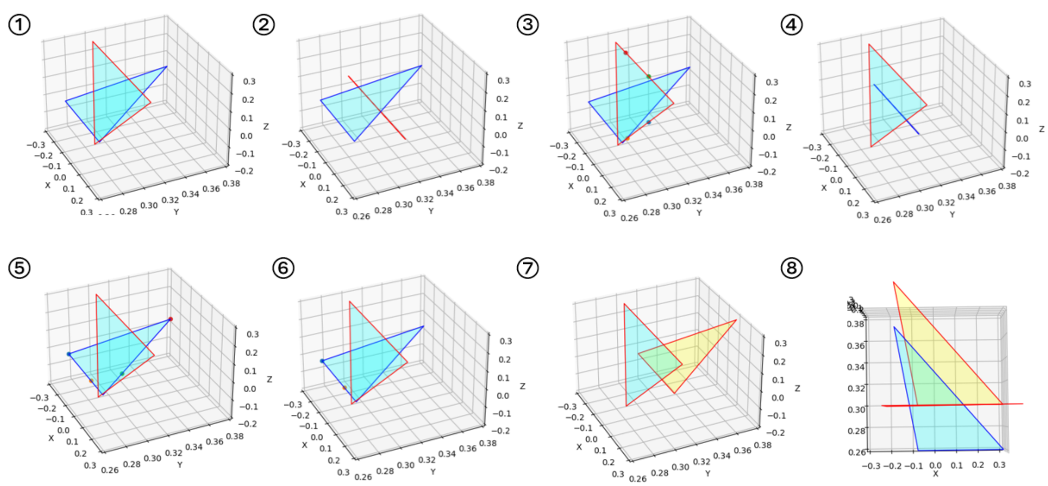

Figure 4: Edge-triangle case when the projected line segment does not fit inside the other triangle.

1.3 Point-Point case

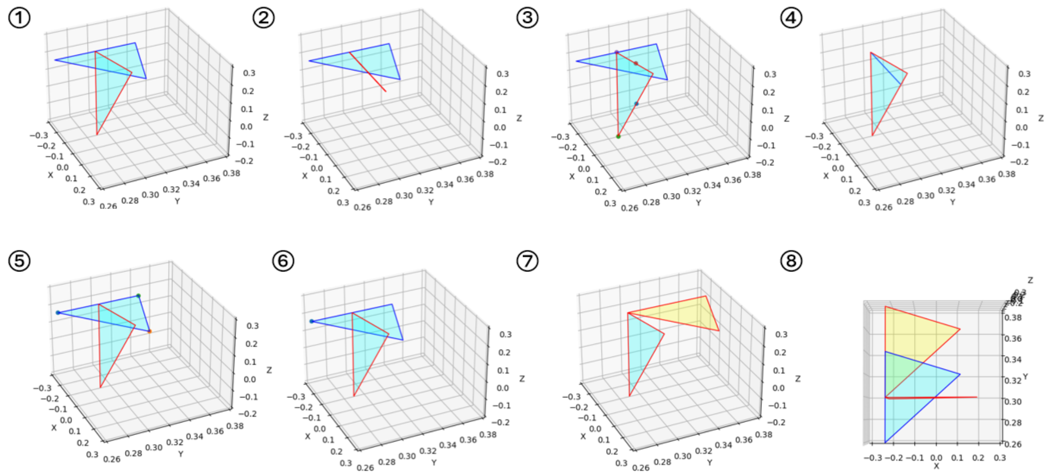

Figure 5: Point-Point case when the projected line segment fits inside the other triangle.

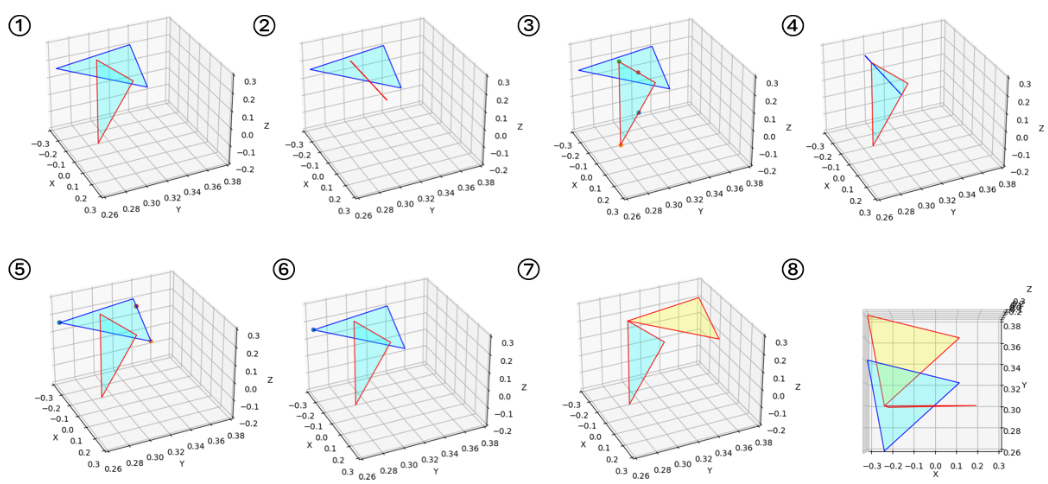

Figure 6: Point-Point case when the projected line segment does not fit inside the other triangle.

1.4 Edge-Edge case

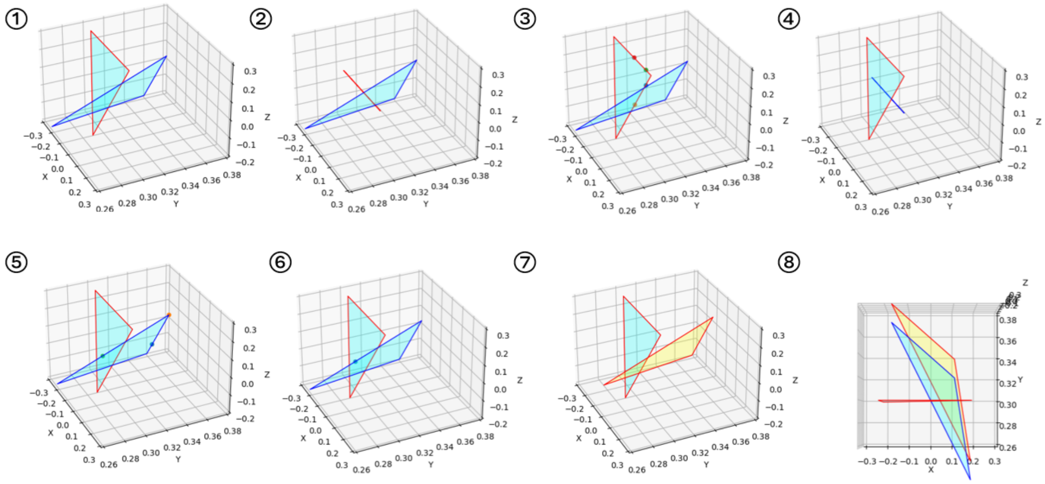

Figure 7: Edge-Edge case

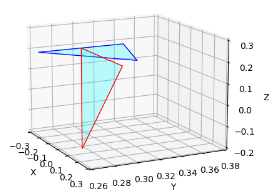

Figure 8: When a projected triangle degenerates onto an edge of the other triangle, it is considered to be in a touching state, and thus it is determined that there is no intersection when intersecting test.

1.5 Point-Edge case

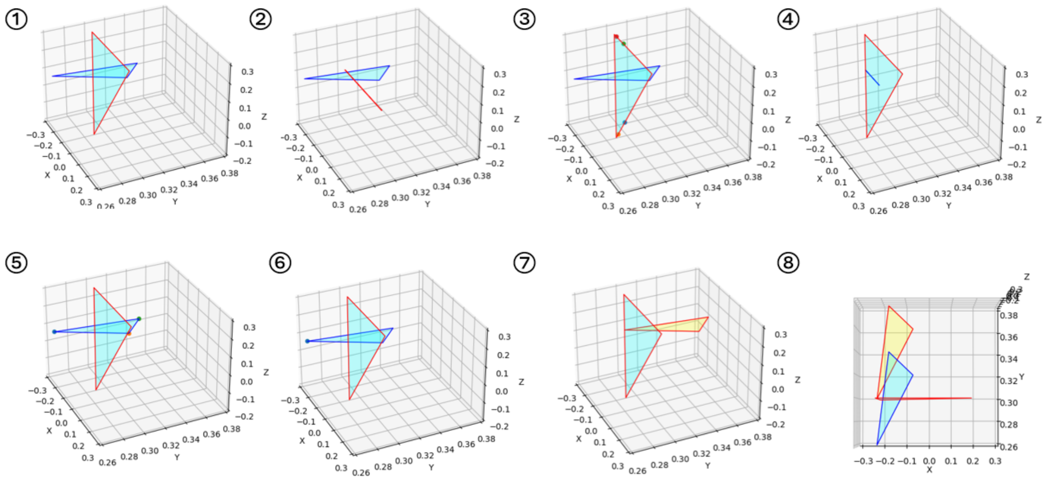

Figure 9: Point-Edge case when the projected line segment fits inside the other triangle.

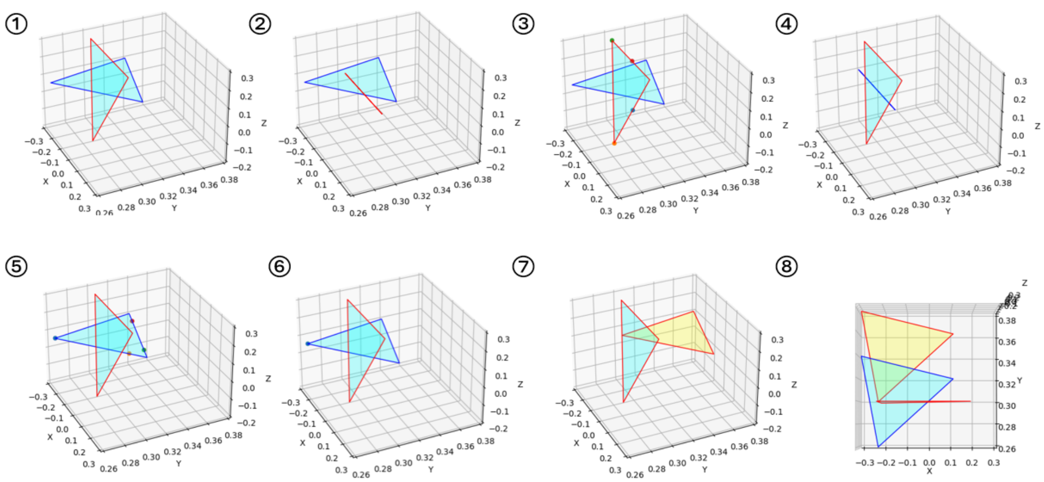

Figure 10: Point-Edge case when the projected line segment does not fit inside the other triangle.

2 Erleben Test

We conducted fundamental collision tests proposed by Erleben. We successfully completed the test without causing breakdowns. There are six cases: Spikes, Spike and wedge, Wedges, Spike in hole, Spike in crack, and Wedge in crack. In some cases, edge collisions occur during the dropping process.

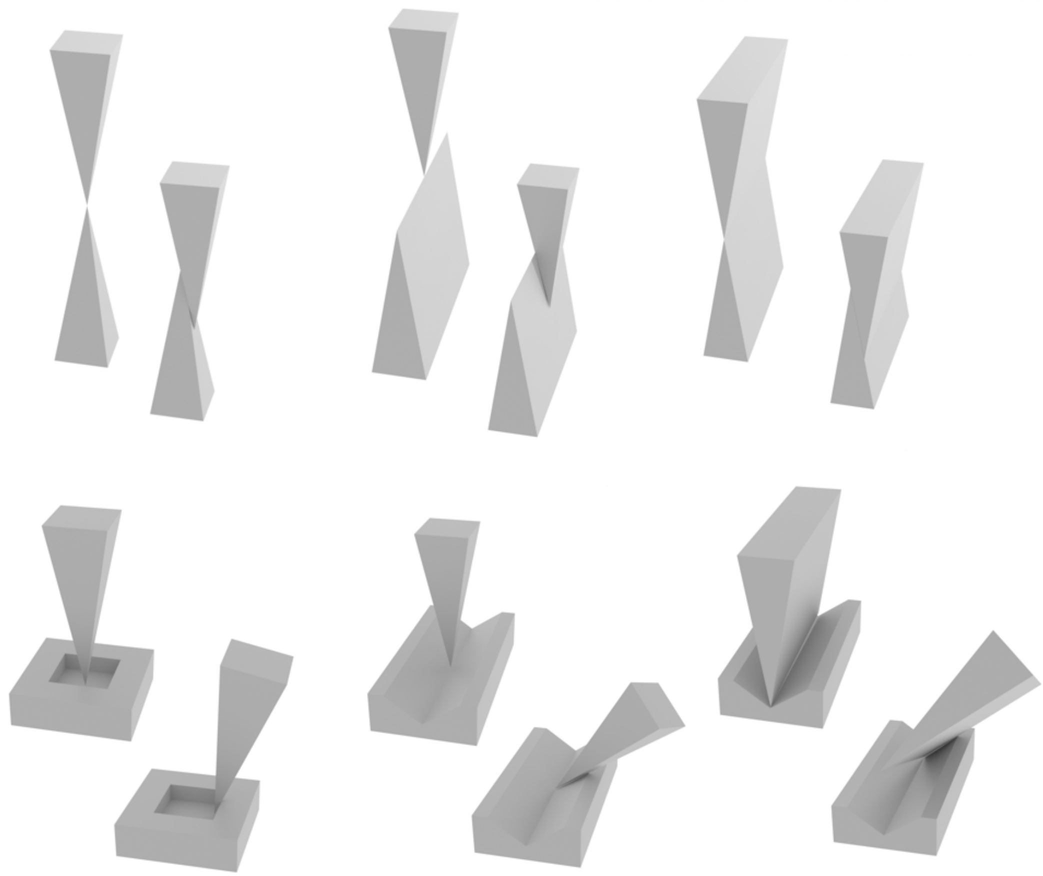

Figure 11: Erleben test results.

3 Derivation of the Gradient $-\boldsymbol{\nabla f}$ from the Function f in Section 3.2 of the paper

In our method, the gradient $-\boldsymbol{\nabla f}$ of the function f corresponds to the normal vector of either one of the triangles. We show the derivation through the following equations.

We begin with Equation 6 of the paper for the function f:

$$\begin{aligned} f &= \min(d_{\text{tri1}}, d_{\text{tri2}}), \\ d_{\text{tri1}} &= \max(0.0, -d_{\text{Q}_l^1}), \\ d_{\text{tri2}} &= \max(0.0, -d_{\text{Q}_l^2}). \end{aligned}$$

Here, f is determined by the minimum of d_{tri1} and d_{tri2}, which are themselves dependent on $d_{\text{Q}_l^1}$ and $d_{\text{Q}_l^2}$. These terms, $d_{\text{Q}_l^1}$ and $d_{\text{Q}_l^2}$, are expressed as:

$$d_{\text{Q}_l^1} = \boldsymbol{N}_2 \cdot \boldsymbol{Q}_l^1 + d_2,$$

and

$$d_{\text{Q}_l^2} = \boldsymbol{N}_1 \cdot \boldsymbol{Q}_l^2 + d_1,$$

where $\boldsymbol{N}_1$ and $\boldsymbol{N}_2$ are the normal vectors corresponding to triangle T_1 and T_2, respectively. In the case where one of the triangles dominates the minimum, the function f takes the form:

$$f = \max(0.0, -(\boldsymbol{N} \cdot \boldsymbol{Q} + d)),$$

where $\boldsymbol{N}$ is the normal vector of the corresponding triangle, and $\boldsymbol{Q}$ is the selected point among the candidates.

Finally, the gradient of f with respect to $\boldsymbol{Q}$ is given by the gradient of the dot product term. Since the gradient of $\boldsymbol{N} \cdot \boldsymbol{Q}$ with respect to $\boldsymbol{Q}$ is $\boldsymbol{N}$, we conclude that:

$$-\boldsymbol{\nabla f} = \boldsymbol{N}.$$

4 Python Code

Here, we provide the Python code for finding the intersecting point on the triangle pair and calculating the PBD constraint C and its gradient $\boldsymbol{\nabla C}$.

The code is all-inclusive, standalone, and includes a snippet of Möller's collision detection method. Out of the 185 lines of code, 65 lines are dedicated to Möller's collision detection, while the remaining 120 lines pertain to the method presented. When using this code in practice, pass the triangle coordinates as follows:

```
triangle1 = np.array([[0.375003, 0.299691, 0.299992],
                      [-0.224997, 0.299691, -0.300008],
                      [-0.224998, 0.299691, 0.299994]])
triangle2 = np.array([[-0.0749846, 0.26, 0.0999057],
                      [0.125025, 0.26, 0.0999057],
                      [-0.1750912, 0.36939, 0.0999057]])
```

Listing 1: Our DCD code

```
import numpy as np

def swap(d0_, d1_, d2_, tri0, tri1, tri2):
    if (d0_ <= 0 and d1_ >= 0 and d2_ >= 0) or (d0_ >= 0 and d1_ <= 0 and d2_ <= 0):
        v0_ = tri1
        v1_ = tri0 # the vertex on the opposite side
        v2_ = tri2
        d_ = d0_
        d0_ = d1_
        d1_ = d_
        d2_ = d2_

    elif (d0_ >= 0 and d1_ <= 0 and d2_ >= 0) or (d0_ <= 0 and d1_ >= 0 and d2_ <= 0):
        v0_ = tri0
        v1_ = tri1 # the vertex on the opposite side
        v2_ = tri2

    elif (d0_ >= 0 and d1_ >= 0 and d2_ <= 0) or (d0_ <= 0 and d1_ <= 0 and d2_ >= 0):
        v0_ = tri0
        v1_ = tri2 # the vertex on the opposite side
        v2_ = tri1
        d_ = d1_
        d0_ = d0_
        d1_ = d2_
        d2_ = d_
    return d0_, d1_, d2_, v0_, v1_, v2_

def swap_minmax(t1_, t2_, d0_, d2_, v0_, v2_):
```

```
    if (t1_ > t2_):
        t_ = t1_
        t1_ = t2_
        t2_ = t_
        d_ = d0_
        d0_ = d2_
        d2_ = d_
        v_ = v0_
        v0_ = v2_
        v2_ = v_
    return t1_, t2_, d0_, d2_, v0_, v2_

def gen_t(N_, tri_, D_, d_):
    e = 0.0 # adjust this threshold value based on the situation (e.g. cloth simulation).
    if d_[0] <= e and d_[1] <= e and d_[2] <= e:
        return None, None
    if d_[0] >= -e and d_[1] >= -e and d_[2] >= -e:
        return None, None
    d0_, d1_, d2_, v0_, v1_, v2_ = swap(d_[0], d_[1], d_[2], tri_[0], tri_[1], tri_[2])
    p0_ = np.dot(D_, v0_)
    p1_ = np.dot(D_, v1_)
    p2_ = np.dot(D_, v2_)

    t1 = p0_ + (p1_ - p0_) * abs(d0_ / (d0_ - d1_))
    t2 = p2_ + (p1_ - p2_) * abs(d2_ / (d2_ - d1_))
    t1, t2, d0_, d2_, v0_, v2_ = swap_minmax(t1, t2, d0_, d2_, v0_, v2_)
    return t1, t2

def line_intersection_on_same_plane(p1, p2, p3, p4, v1, v2):
    d1 = p2 - p1
    d2 = p4 - p3
    n = np.cross(d1, d2)
    if np.linalg.norm(n) == 0:
        return None
    denom = np.dot(n, n)
    if denom == 0:
        return None
    v = p3 - p1
    t1 = np.dot(np.cross(v, d2), n) / denom
    t2 = np.dot(np.cross(v, d1), n) / denom
    if (0 <= t2 and t2 <= 1) and (0 <= t1 and t1 <= 1):
        return v1 + t1 * (v2 - v1)
    return None

def inside_triangle_on_same_plane(triangle, p):
```

```
    ab = triangle[1] - triangle[0]
    bp = p - triangle[1]

    bc = triangle[2] - triangle[1]
    cp = p - triangle[2]

    ca = triangle[0] - triangle[2]
    ap = p - triangle[0]

    c1 = np.cross(ab, bp)
    c2 = np.cross(bc, cp)
    c3 = np.cross(ca, ap)

    if (np.dot(c1, c2) > 0 and np.dot(c1, c3) > 0):
        return True
    return False

def find_intersection_point(N, p0, p, line_dir):
    if (np.dot(N, line_dir) == 0.0):
        return None
    t = np.dot(N, p0 - p) / np.dot(N, line_dir)
    intersection_point = p + t * line_dir
    return intersection_point

#Eq. 1
N1 = np.cross(triangle1[1] - triangle1[0], triangle1[2] -
    triangle1[0])
N1 = N1 / np.linalg.norm(N1)
d1 = -np.dot(N1, triangle1[0])
N2 = np.cross(triangle2[1] - triangle2[0], triangle2[2] -
    triangle2[0])
N2 = N2 / np.linalg.norm(N2)
d2 = -np.dot(N2, triangle2[0])

d_on_vertex = np.zeros((2, 3, 1))
for i in range(3):
    d_on_vertex[0][i] = np.dot(N2, triangle1[i]) + d2
for i in range(3):
    d_on_vertex[1][i] = np.dot(N1, triangle2[i]) + d1

# Section 3.1 Moller's intersecting test
D = np.cross(N1, N2)
D = D / np.linalg.norm(D)
t1, t2 = gen_t(N2, triangle1, D, d_on_vertex[0])
if (t1 is None):
    exit()
t3, t4 = gen_t(N1, triangle2, D, d_on_vertex[1])
if (t3 is None):
```

```
    exit()
if not (t2 >= t3 and t4 >= t1):
    exit()

# Section 3.2
# Eq. 4
P = np.zeros((2, 3, 3))
for i in range(3):
    P[0][i] = triangle1[i] - np.dot(N2, triangle1[i] - triangle2[0]) * N2
for i in range(3):
    P[1][i] = triangle2[i] - np.dot(N1, triangle2[i] - triangle1[0]) * N1

# Eq. 5
Q = np.zeros((2, 3, 3))
for i in range(3):
    Q[0][i] = find_intersection_point(N1, triangle1[0], triangle2[i], N2)
for i in range(3):
    Q[1][i] = find_intersection_point(N2, triangle2[0], triangle1[i], N1)

# Append the candidates of each triangle
intersections1 = []
d_tri1 = 0.0
d_tri2 = 0.0
for i in range(3):
    if (inside_triangle_on_same_plane(triangle2, P[0][i])):
        if (d_tri1 < -d_on_vertex[0][i]):
            d_tri1 = -d_on_vertex[0][i]
    if (Q[0][i] is not None):
        if (inside_triangle_on_same_plane(triangle1, Q[0][i])):
            intersections1.append(Q[0][i])
    for j in range(3):
        intersection = line_intersection_on_same_plane(P[0][i], P[0][(i+1)%3], triangle2[j], triangle2[(j+1)%3], triangle1[i], triangle1[(i+1)%3])
        if intersection is not None:
            intersections1.append(intersection)

intersections2 = []
for i in range(3):
    if (inside_triangle_on_same_plane(triangle1, P[1][i])):
        if (d_tri2 < -d_on_vertex[1][i]):
            d_tri2 = -d_on_vertex[1][i]
    if (Q[1][i] is not None):
        if (inside_triangle_on_same_plane(triangle2, Q[1][i])):
```

```
            intersections2.append(Q[1][i])
    for j in range(3):
        intersection = line_intersection_on_same_plane(P[1][i], P[1][(i+1)%3], triangle1[j], triangle1[(j+1)%3], triangle2[i], triangle2[(i+1)%3])
        if intersection is not None:
            intersections2.append(intersection)

for v in intersections1:
    candidate = np.dot(N2, v) + d2
    if (d_tri1 < -candidate):
        d_tri1 = -candidate

for v in intersections2:
    candidate = np.dot(N1, v) + d1
    if (d_tri2 < -candidate):
        d_tri2 = -candidate

if (d_tri1 > 1.0e-7 and d_tri1 < d_tri2):
    C = d_tri1
    dC = N2 # Define dC as the direction in which C decreases. Mathematically, -dC = N2.
if (d_tri2 > 1.0e-7 and d_tri2 < d_tri1):
    C = d_tri2
    dC = N1 # Define dC as the direction in which C decreases. Mathematically, -dC = N1.

print(C, dC)
```