Growth distance implementation

In [12]:
```
#Tetrahedron subdivision
import numpy as np

odf_cartesian_faces_dict = {}  # Dictionary to store tetrahedrons for each ODF center

for i in range(odf.shape[0]):
    for j in range(odf.shape[1]):
        for k in range(odf.shape[2]):
            voxel_center = np.array([i, j, k])  # Center of the current voxel

            for n in range(odf.shape[3]):
                voxel_faces = []  # List to store Cartesian coordinates for each face in the voxel
                face_indices = sphere.faces[n]  # Face indices corresponding to the current direction

                face_vertices = [voxel_center]  # Add the center of the voxel as the first vertex

                for idx in face_indices:
                    cartesian_coords = voxel_center + odf[i, j, k, n] * sphere.vertices[idx]

                    face_vertices.append(cartesian_coords)

                voxel_faces.append(face_vertices)

                # Add tetrahedrons to the corresponding ODF center in the dictionary
                odf_center_key = tuple(voxel_center)  # Use the voxel_center as the key
                if odf_center_key not in odf_cartesian_faces_dict:
                    odf_cartesian_faces_dict[odf_center_key] = []
                odf_cartesian_faces_dict[odf_center_key].extend(voxel_faces)
# odf_cartesian_faces_dict contains tetrahedrons for each ODF center
```

In [ ]:
```
#the tetrahedrons of the first ODF with center [0, 0, 0]
odf_cartesian_faces_dict[(0, 0, 0)]
```

In [39]:
```
#Implementing the growth distance algorithm
Aeq = np.column_stack(A)
ones_row = np.ones((1, Aeq.shape[1]))
Aeq = np.concatenate((Aeq, ones_row), axis=0)
Aeq
```

Out[39]:
```
array([[ 0.00000000e+00, -4.45509482e-05,  1.55800182e-05,
         3.20443748e-06],
       [ 0.00000000e+00,  4.84966050e-04,  4.86325202e-04,
         4.84314000e-04],
       [ 0.00000000e+00,  7.44070161e-06, -2.18467780e-05,
         5.15936194e-05],
       [ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
         1.00000000e+00]])
```

In [41]:
```
Beq = np.column_stack(B)
minus_ones_row = np.ones((1, Beq.shape[1])) * -1
Beq = np.concatenate((Beq, minus_ones_row), axis=0)
Beq
```

Out[41]:
```
array([[ 0.00000000e+00, -2.38126082e-04, -2.98936667e-04,
        -3.11225567e-04],
       [ 0.00000000e+00, -6.00621932e-04, -5.74394913e-04,
        -5.62277423e-04],
       [ 1.00000000e+00,  1.00004530e+00,  9.99985505e-01,
         1.00008053e+00],
       [-1.00000000e+00, -1.00000000e+00, -1.00000000e+00,
        -1.00000000e+00]])
```

In [43]:
```
combined_matrix = np.hstack((Aeq, Beq))
combined_matrix
```

Out[43]:
```
array([[ 0.00000000e+00, -4.45509482e-05,  1.55800182e-05,
         3.20443748e-06,  0.00000000e+00, -2.38126082e-04,
        -2.98936667e-04, -3.11225567e-04],
       [ 0.00000000e+00,  4.84966050e-04,  4.86325202e-04,
         4.84314000e-04,  0.00000000e+00, -6.00621932e-04,
        -5.74394913e-04, -5.62277423e-04],
       [ 0.00000000e+00,  7.44070161e-06, -2.18467780e-05,
         5.15936194e-05,  1.00000000e+00,  1.00004530e+00,
         9.99985505e-01,  1.00008053e+00],
       [ 1.00000000e+00,  1.00000000e+00,  1.00000000e+00,
         1.00000000e+00, -1.00000000e+00, -1.00000000e+00,
        -1.00000000e+00, -1.00000000e+00]])
```

In [50]:
```
pB_minus_pA = B[0] - A[0]
pB_minus_pA
```

Out[50]: array([0, 0, 1])

In [51]:
```
b = np.concatenate((pB_minus_pA, [0]))
b
```

Out[51]: array([0, 0, 1, 0])

In [31]:
```
#Exemple of the growth distance between 2 tetrahedrons of 2 different fodfs
import numpy as np
from scipy.spatial import ConvexHull
from scipy.optimize import linprog

def calculate_convex_hull(vertices):
    hull = ConvexHull(vertices)
    return np.array(vertices)[hull.vertices]

def solve_lp_problem(bar_A, bar_B):

    num_A = len(bar_A)
    num_B = len(bar_B)

    # Construct the constraint matrix combined_matrix
    Aeq = np.column_stack(bar_A)
    ones_row = np.ones((1, Aeq.shape[1]))
    Aeq = np.concatenate((Aeq, ones_row), axis=0)

    Beq = np.column_stack(bar_B)
    minus_ones_row = np.ones((1, Beq.shape[1])) * -1
    Beq = np.concatenate((Beq, minus_ones_row), axis=0)

    combined_matrix = np.hstack((Aeq, Beq))
    # Coefficients of the objective function
    c = np.ones(num_A+num_B)

    pB_minus_pA = bar_B[0] - bar_A[0]

    # Construct the right-hand side vector b_eq
    b_eq = np.concatenate((pB_minus_pA, [0]))

    bounds = [(0, None)] * (num_A + num_B)  # Bounds for alpha_A and alpha_B

    res = linprog(c, A_eq=combined_matrix, b_eq=b_eq, bounds=bounds)
    sigma_star = res.fun
    alpha_A = res.x[:num_A]
    alpha_B = res.x[num_A:]
    return sigma_star, alpha_A, alpha_B
```

In [33]:
```
# Calculate convex hulls of A and B
A = odf_cartesian_faces[0]   #tetrahedron1
B = odf_cartesian_faces[800]   #tetrahedron2
bar_A = calculate_convex_hull(A)
bar_B = calculate_convex_hull(B)


sigma_star, alpha_A, alpha_B = solve_lp_problem(bar_A, bar_B)

print("sigma_star:", sigma_star)
print("alpha_A:", alpha_A)
print("alpha_B:", alpha_B)
```

```
sigma_star: 2.0000000035223438
alpha_A: [1.00000000e+00 8.50020595e-10 5.04347761e-10 5.87846481e-10]
alpha_B: [1.00000000e+00 1.33106094e-09 1.31830763e-10 4.13307623e-10]
```

The growth distance is given by the factor sigma star = 2.0000000035223438

In [17]:
```
#Exemple of the growth distance between 2 different fodfs
# Retrieve the tetrahedrons for ODF centers (0, 0, 0) and (0, 0, 1)
tetrahedrons_A = odf_cartesian_faces_dict[(0, 0, 0)]
tetrahedrons_B = odf_cartesian_faces_dict[(0, 0, 1)]
```

In [24]:
```python
import numpy as np
from scipy.spatial import ConvexHull
from scipy.optimize import linprog
import time

def calculate_convex_hull(vertices):
    hull = ConvexHull(vertices)
    return np.array(vertices)[hull.vertices]

def solve_lp_problem(bar_A, bar_B):
    num_A = len(bar_A)
    num_B = len(bar_B)

    # Construct the constraint matrix A_eq
    Aeq = np.column_stack(bar_A)
    ones_row = np.ones((1, Aeq.shape[1]))
    Aeq = np.concatenate((Aeq, ones_row), axis=0)

    Beq = np.column_stack(bar_B)
    minus_ones_row = np.ones((1, Beq.shape[1])) * -1
    Beq = np.concatenate((Beq, minus_ones_row), axis=0)

    combined_matrix = np.hstack((Aeq, Beq))
    # Coefficients of the objective function
    c = np.ones(num_A + num_B)

    pB_minus_pA = bar_B[0] - bar_A[0]

    # Construct the right-hand side vector b_eq
    b_eq = np.concatenate((pB_minus_pA, [0]))

    bounds = [(0, None)] * (num_A + num_B)  # Bounds for alpha_A and alpha_B

    res = linprog(c, A_eq=combined_matrix, b_eq=b_eq, bounds=bounds)
    sigma_star = res.fun
    alpha_A = res.x[:num_A]
    alpha_B = res.x[num_A:]
    return sigma_star
```

In [ ]:
```python
def calculate_optimized_sigma_star(tetrahedrons_A, tetrahedrons_B):
    optimized_sigma_star = float('inf')  # Initialize with a large value

    # Loop over the tetrahedrons of ODF centers (0, 0, 0) and (0, 0, 1)
    for tetrahedron_A in tetrahedrons_A:
        for tetrahedron_B in tetrahedrons_B:
            # Calculate the convex hulls for each pair of tetrahedrons
            convex_hull_A = calculate_convex_hull(tetrahedron_A)
            convex_hull_B = calculate_convex_hull(tetrahedron_B)

            # Solve the linear programming problem between the convex hulls
            sigma_star = solve_lp_problem(convex_hull_A, convex_hull_B)

            # Update the optimized_sigma_star if a smaller value is found
            optimized_sigma_star = min(optimized_sigma_star, sigma_star)

    return optimized_sigma_star
```

In [ ]:
```python
def run_optimization(tetrahedrons_A, tetrahedrons_B):
    start_time = time.time()
    optimized_sigma_star = calculate_optimized_sigma_star(tetrahedrons_A, tetrahedrons_B)
    end_time = time.time()
    runtime_seconds = end_time - start_time
    runtime_minutes = runtime_seconds / 60
    # Print runtime in minutes
    print("Runtime of the code is:", runtime_minutes, "minutes.")

    # Print the optimized sigma_star
    print("Optimized sigma_star:", optimized_sigma_star)
```

In [ ]:
```python
run_optimization(tetrahedrons_A, tetrahedrons_B)
```

Runtime of the code is: 9.171941713492076 minutes. Optimized sigma_star: 1.9809936553587417

The optimized sigma_star value of 1.9809936553587417 indicates the minimum scaling factor needed for the two FODFs to intersect, based on the linear programming solution. This value plays a crucial role in the computation of the growth distance between two FODFs, which is a key step in the topological data analysis process.

As for the runtime, 9 minutes per pair of FODFs may be time-consuming, especially since we are dealing with a large number of FODFs.