Math574 - Homework3

Altansuren Tumurbaatar

1.(20) Find the Euler characteristic of the given 2-complex. Explain how you are finding the (list of) vertices and edges (so as to count the numbers of them).

List of edges:

For each triangle

For each edge of the triangle

Order the edge lexicographically

Add the edge to a set of edges

Length of the edges list is the number of edges

List of vertices:

Get all distinct vertices from the List of edges

Euler characteristic

```
#Euler characteristic
def kai(simplices):
    '''
        Subtracting subsimplices starting from the highest order simplices
```

---

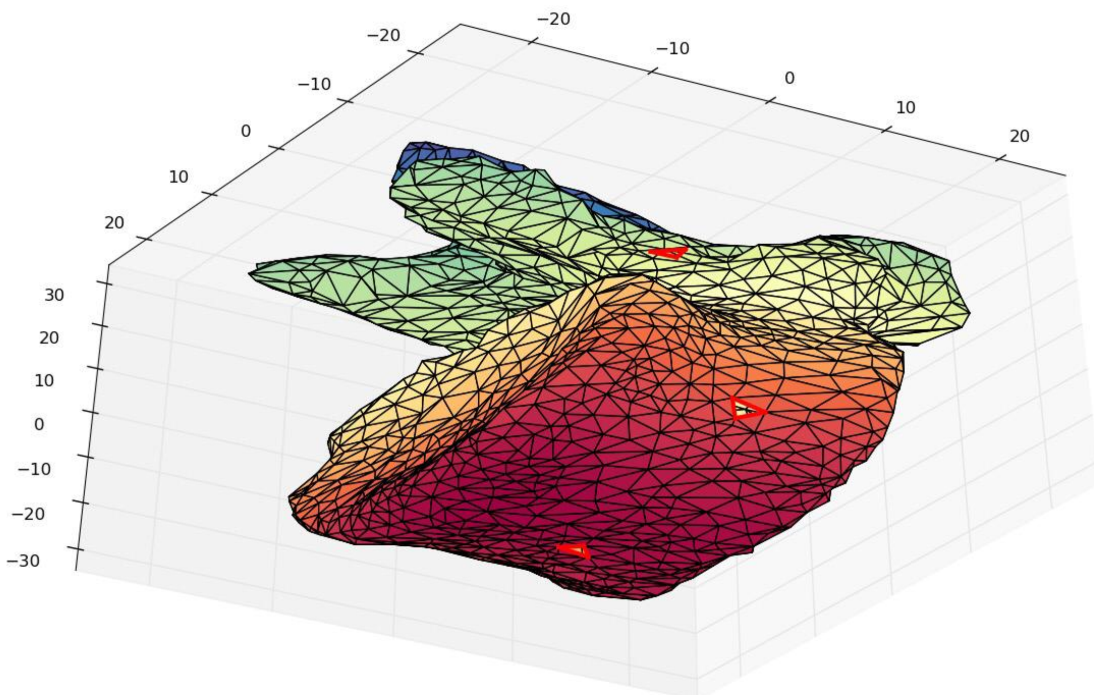
surface.py

```
        and counting their numbers.
        Sign of the one term is decided by its dimension.
        For example, X(K) = V - E + F -T so on.
    '''
    X = 0
    while simplices.shape[1] >= 1:
        X += ((-1)**(simplices.shape[1]-1))*simplices.shape[0]
        simplices = get_subsimplices(simplices)
    return X
```

2. (30) Is the 2-complex a surface? If not, could you make it a surface by adding a few missing triangles?

- There is no edge shared by 3 triangles and also all the edges are face of at least one triangle.
- Euler characteristic of the 2-complex is -3.

- It is not a 2-manifold, but it is a 2-manifold with boundary that there are 6 edges shared by only one triangles. Thus the 6 edges are boundaries.
- 6-edges constitutes boundary of 3 missing triangles. When we fill the triangles by adding them to the list of triangles. The 2-complex becomes a surface without any hole or 2-manifold without boundary.



- N0_points – Isolated points which are discarded later

  N0_edges – Edges shared by no triangle.

  N1_edges – Edges shared by one triangle

  N3_edges – Edges shared by more than 2 triangles

```
Euler Characteristic
-3
=================================================================
Neighborhood information(N1-edges shared by only one triangle)
N1_edges
[499 502]
[ 342 1257]
[500 502]
[499 500]
[91 95]
[ 95 264]
[ 349 1257]
[342 349]
[ 91 264]
N0_points [ 269  508 1896 1925]
N3_edges []
N0_edges []
=================================================================
Missing triangles below are added:
[499, 500, 502]
[342, 349, 1257]
[91, 95, 264]
```

3.(50) Assuming you have a surface from the previous step, can you orient the surface? You could try to propagate a chosen orientation from a single triangle to all other triangles. What standard surface, if any, is this one homeomorphic to?

True. We can orient the surface by propagating induced orientation starting from a random triangle that induced orientations(from triangles intersecting the edge) on each edge need to be opposite where each edge is part of two triangles, so each edge edge have two induced orientations from the two triangle.

1. Create boundary matrix without any orientation with 0,1,-1 entries

    and use it to filter adjacent triangles and edges of a triangle

2. Add all triangles to all_tris list

3. Start with 0th triangle

4. Propagate orientation to adjacent triangles:

        for each edge, find adjacent triangle and compute induced orientation of the edge on it.

            if adjacent triangle is not in oriented_tris

                if induced orientations on the shared edge doesn't cancel each other,

                do one swap on adjacent triangle and

                put corresponding orientation entry current triangle

                and edge in the boundary matrix

                Add current and adjacent triangles to oriented_tris set

            if adjacent triangle is in oriented_tris

                check if the induced orientations on the shared edge are opposit,

                    then put the orientation entry in the boundary matrix position

                    [edge, adj_traingle]

                if not, return False

5. Remove current triangle from all_tris

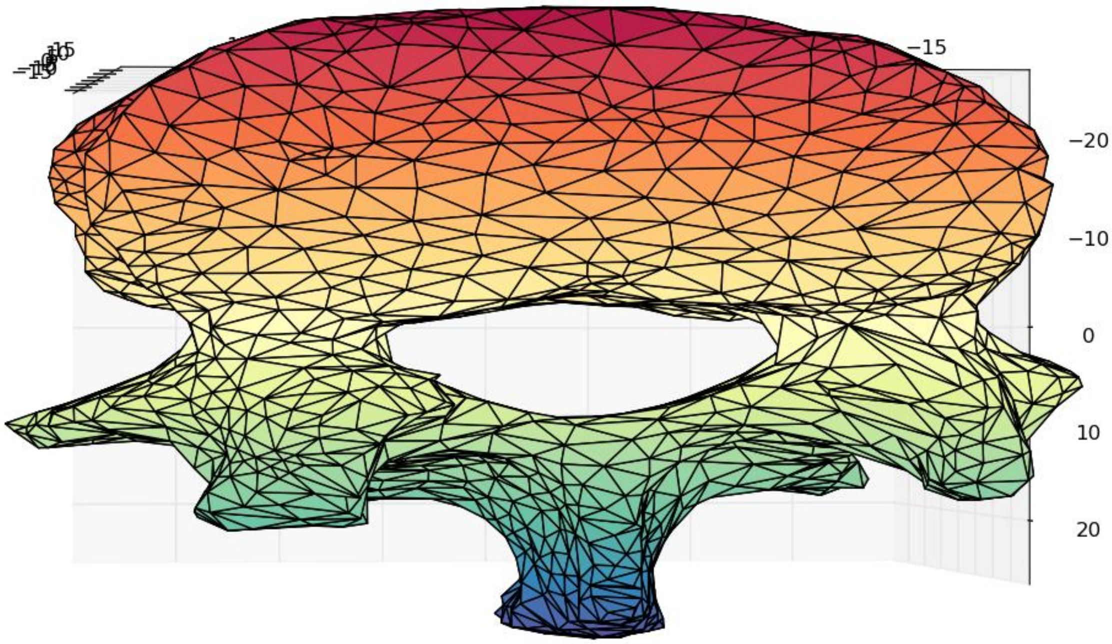
6. Repeat 4 till all_tris list is empty(so all the triangles are checked)

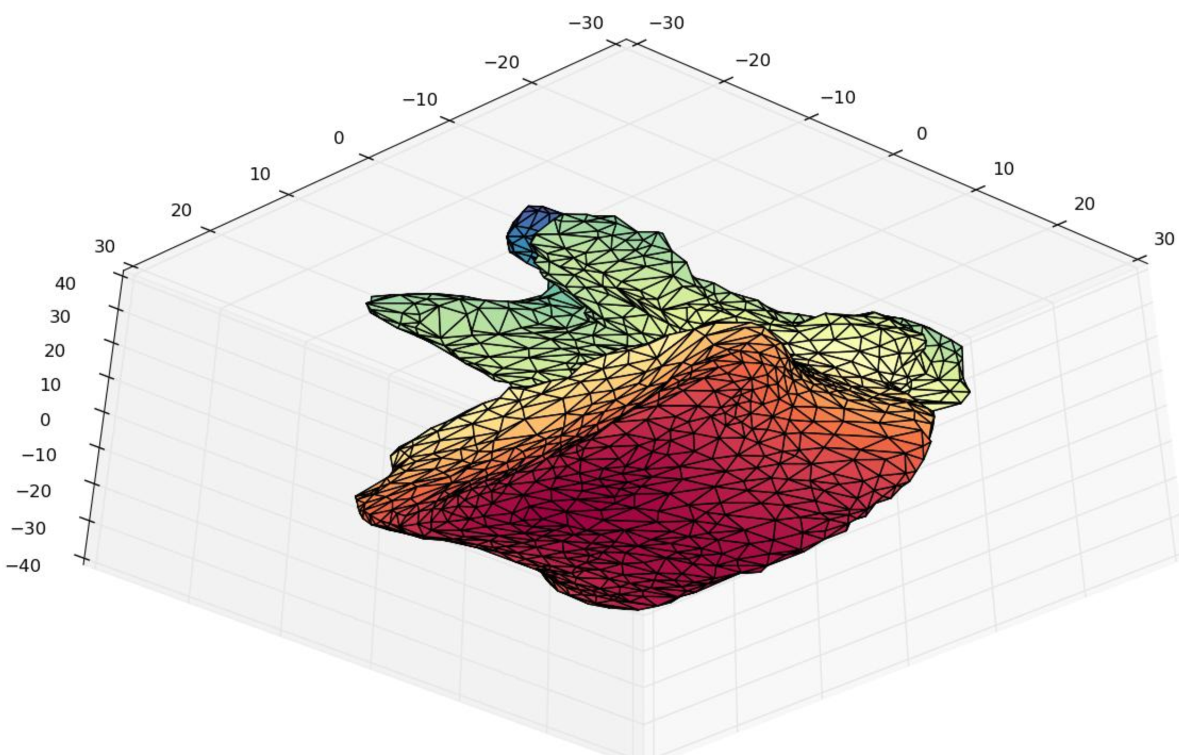
7. Return True

The surface is orientable and has Euler characteristic, 0 which is a torus since orientiability and Euler characteristic together constitutes a complete topological invariant.

```
===========================================================================
Euler characteristic
0
===========================================================================
Orientable:
True
```

4.(30) Produce a few views of the patched up surface using TetView, or another meshing software (or in Octave or Python). Include these views as images in your report.





```
'''
Math574 - Computationa Topology - Spring 2016
---------------------------------------------

Homework3 on analysing surfaces.
Triagles.txt - List of triangles(mesh) which approxiates the surface
Vertices.txt - Coordinates of the points used in the trianglular mesh

*Tasks:*
.# Compute Euler characteristic
.# Is the 2-complex surface? Missing triangles to complete it as a surface
.# Orient the 2-complex and identify if it is homemorphic to any standard surfaces
.# Plot the surface

'''

import numpy as np
from scipy.sparse import dok_matrix

from mpl_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt

def get_subsimplices(simplices):
    simplices = np.sort(simplices, axis=1)
    subsimplices = set()
    for j in np.arange(simplices.shape[1]):
        idx = list(range(simplices.shape[1]))
        idx.pop(j)
        subsimplices = subsimplices.union(set(tuple(sub) for sub in simplices.take(idx
, axis=1)))
    subsimplices = np.array([ sub for sub in subsimplices], dtype=int)
    return subsimplices

def boundary_matrix(simplices, subsimplices, is_sparse=True, format='coo'):
    simplex_dim  = simplices.shape[1]
    n_simplices = simplices.shape[0]
    m_subsimplices = subsimplices.shape[0]
    if is_sparse:
        boundary_matrix = dok_matrix((m_subsimplices, n_simplices), dtype=np.int8)
    else:
        boundary_matrix = np.array((m_subsimplices, n_simplices), dtype=np.int8)
    val = 1
    simplices = np.sort(simplices, axis=1)
    subsimplices = np.sort(subsimplices, axis=1)
    for i, simplex in enumerate(simplices):
        for j in np.arange(simplex_dim):
            idx = list(range(simplex_dim))
            idx.pop(j)
            subsimplex = simplex.take(idx)
            # to check the membership of subsimplex in subsimplices
            subsimplex_idx = np.argwhere((subsimplices==subsimplex).all(axis=1) == Tru
e)
            if subsimplex_idx.size == 0:
                sys.stderr.write("Unable to find subsimplex! Make sure subsimplices co
ntains all boundary subsimplices\n")
                exit()
            subsimplex_idx = subsimplex_idx[0][0]
            boundary_matrix[subsimplex_idx, i] = val
    if is_sparse:
        return boundary_matrix.asformat(format)
    else:
        return boundary_matrix


#Euler characteristic
def kai(simplices):
    '''
        Subtracting subsimplices starting from the highest order simplices
```

```
        and counting their numbers.
        Sign of the one term is decided by its dimension.
        For example, X(K) = V - E + F -T so on.
    '''
    X = 0
    while simplices.shape[1] >= 1:
        X += ((-1)**(simplices.shape[1]-1))*simplices.shape[0]
        simplices = get_subsimplices(simplices)
    return X

def check_surface(triangles, edges, no_of_points):
    '''
    Identifies if the triangular mesh represents a surface.
    Check:
     * if there is an edge which is shared by more than one or two triangles
     * if there is an edge which is not a face of any triangle
     * if there is an isolated point which is not a part of any edge
    '''
    exceptions = {"N0_points":[], "N0_edges":[],"N1_edges":[], "N3_edges":[]}
    boundary_1 = boundary_matrix(edges, np.arange(no_of_points).reshape(-1,1), format=
"dok")
    boundary_2 = boundary_matrix(triangles, edges, format="dok")
    N0_points = np.where(boundary_1.sum(axis=1)==0)[0].tolist()[0]
    N0_edges = np.where(boundary_2.sum(axis=1)==0)[0].tolist()[0]
    N1_edges = np.where(boundary_2.sum(axis=1)==1)[0].tolist()[0]
    N3_edges = np.where(boundary_2.sum(axis=1) > 2)[0].tolist()[0]
    is_surface = True
    if len(N0_points) != 0:
        exceptions["N0_points"] = N0_points
        is_surface = False
    if len(N0_edges) != 0:
        exceptions["N0_edges"] = N0_edges
        is_surface = False
    if len(N1_edges) !=0:
        exceptions["N1_edges"] = N1_edges
    if len(N3_edges) != 0:
        exceptions["N3_edges"] = N3_edges
        is_surface = False
    return is_surface, exceptions

def edge_sign(simplex, edge):
    '''
    Given a triangle and its edge, return the induced orientation
    of the edge.
    '''
    for j in np.arange(3):
        idx = list(range(3))
        idx.pop(j)
        e = simplex.take(idx)
        flag = (-1)**j
        if flag == -1:
            e = e[::-1]
        if np.all(e == edge):
            return 1
        elif np.all(e==edge[::-1]):
            return -1

def check_orientation(simplices,edges):
    '''
        1. Create boundary matrix without any orientation with 0,1,-1 entries
           and use it to filter adjacent triangles and edges of a triangle
        2. Add all triangles to all_tris list
        3. Start with 0th triangle
        4. Propograte orientation to adjacent triangles:
           for each edge, find adjacent triangle and compute induced orientation of t
he edge on it.
              if adjacent triangle is not in oriented_tris
                 if induced orientations on the shared edge doesn't cancel each oth
er,
```

```
                        do one swap on adjacent triangle and
                        put corresponding orientation entry current triangel
                        and edge in the boundary matrix
                    Add current and adjacent triangles to oriented_tris set

                if adjacent triangle is in oriented_tris
                    check if the induced orientations on the shared edge are opposit,
                    then put the orientation entry in the boundary matrix position [ed
ge, adj_traingle]
                    if not, return False
        5. Remove current triangle from all_tris
        6. Repeat 4 till all_tris list is empty(so all the triangles are checked)
        7. Return True
    '''
    triangles = simplices.copy()
    boundary_2 = boundary_matrix(triangles, edges, format="dok")
    all_tris = range(len(triangles))
    oriented_tris = set()
    tris = list()
    tris.append(0)
    while len(all_tris) != 0:
        t_idx = tris.pop(0)
        all_tris.remove(t_idx)
        print len(all_tris)
        oriented_tris.add(t_idx)
        triangle = triangles[t_idx]
        e_indices =  [ int(k[0]) for k, v in boundary_2.getcol(t_idx).items()]
        for edge_idx in e_indices:
            edge = edges[edge_idx]
            flag1 = edge_sign(triangle, edge)
            boundary_2[edge_idx, t_idx] = flag1
            adj_t =  [ int(k[1]) for k, v in boundary_2.getrow(edge_idx).items() if k[
1] != t_idx][0]
            flag2 = edge_sign(triangles[adj_t], edge)
            if adj_t not in oriented_tris:
                if flag2 == flag1:
                    triangles[adj_t] = triangles[adj_t, [1,0,2]]
                    boundary_2[edge_idx,adj_t] = -flag1
                oriented_tris.add(adj_t)
                tris.append(adj_t)

            else:
                if flag2 == flag1:
                    return False
                else:
                    boundary_2[edge_idx,adj_t] = flag2
    return True

def surface():
    '''
    Findings:
    * There are 4 stand alone vertices and no edges shared by more that two or no tria
ngles.
    * There are 3 triangles missing. On the other words, 6 edges intersecting only one
 triangle.
    * After filling the missing triangles and cheching orientability, calculated Euler
 characteristic
    * The surface is orientable and has Euler characteristic X(K) = 0. The facts imply
 that it is
     2-manifold homeomorphic to a torus.
    '''
    triangles = np.loadtxt("Triangles.txt")
    triangles = triangles -1
    points = np.loadtxt("Vertices.txt")
    X = kai(triangles)
    edges = get_subsimplices(triangles)
    is_surface, exceptions = check_surface(triangles, edges, len(points))
    print "============================================================="
    print "Euler Characteristic"
```

```
    print X
    print "================================================================="
    print "Neighborhood information(N1-edges shared by only one triangle)"
    for k, v in exceptions.items():
        if k=="N1_edges":
            print k
            for i, e in enumerate(exceptions["N1_edges"]):
                print edges[e]+1
        else:
            print k, np.array(v)+1
    N0_points = points[exceptions["N0_points"]]
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    ax.plot_trisurf(points[:,0], points[:,1], points[:,2], triangles=triangles, cmap=p
lt.cm.Spectral)
    #ax.scatter(N0_points[:,0], N0_points[:,1], N0_points[:,2], color="red", s=200)
    for e in exceptions["N1_edges"]:
        edge = points[edges[e]]
        ax.plot(edge[:,0], edge[:,1], edge[:,2], color="red", linewidth="3")
    plt.show()
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    print "================================================================="
    print "Missing triangles below are added:"
    print [499, 500, 502]
    print [342, 349, 1257]
    print [91,95,264]
    #Add missing triangles
    triangles = np.vstack((triangles, np.array([498,499,501]).reshape(1,-1)))
    triangles = np.vstack((triangles, np.array([341,348,1256]).reshape(1,-1)))
    triangles = np.vstack((triangles, np.array([90,94,263]).reshape(1,-1)))
    ax.plot_trisurf(points[:,0], points[:,1], points[:,2], triangles=triangles, cmap=p
lt.cm.Spectral)
    plt.show()
    print "================================================================="
    print "Euler characteristic"
    print kai(triangles)
    print "================================================================="
    print "Orientable:"
    print check_orientation(triangles, edges)

if __name__ == "__main__":
    surface();
```