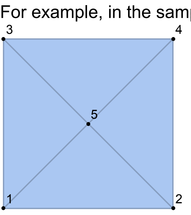
As program setup, we start with generate N random points. For convenience of debugging a default set of 5 points is provided to us.

We compute the Delaunay Mesh for the given set of generated point using “DelaunayMesh” function and store it in a variable. We triangulate the mesh generated (i.e. create triangles on the mesh) using the “MeshCells” function. Thus we now have a mesh of triangles created on N random points.

Part I:

In Part I, we were supposed to find the neighbor list for the given triangulation. A neighbor list is nothing but a matrix in which each row corresponds to a triangle in the mesh. The index of the row gives the triangle in consideration. Thus, if the number of triangle in the mesh is “numtris”, the neighbor list is a (numtris x 3) triangle. The three values in a row in neighbor list corresponds to the triangle number of the triangle opposite to the vertex with that index in the triangle list “tris”.

For example, in the sample mesh,



Triangle 1 in the triangle list “tris”, corresponds to the triangle, {3,1,5} . Hence, the 1st, row of the neighbor list will tell us the neighbor of Triangle 1. As the first vertex in the triangle is 3, the first value at 1st index for the 1st row in neighbor list will be triangle opposite to the vertex 3. This triangle happens to be Triangle {5,1,2} which is Triangle 3 in our triangle list. Thus the 1st value in 1st row of neighbor list will be 3. Similarly, for vertex 1, it is 4, corresponding to the triangle {5,4,3}. We now notice, that there may be some vertices which do not have a triangle facing them, such as vertex 5. We assign a default value of 0 to such vertices.

Hence, the 1st row of neighbor list for this triangulation comes out to be {3,4,0}. In this way we can create the entire neighbor list for all triangles in the mesh.

Now, to compute it, we have used the property that the triangle opposite a vertex will have an edge common to the triangle in which the vertex is. Thus, of all triangles in the mesh, there will be only one triangle which will share the other two vertices of the triangle. Hence, that triangle is the neighbor of that vertex. For example, for vertex 3 in triangle 1, vertices 1 & 5 are shared by triangle 1 and 3 simultaneously. Thus, we assign triangle 3 as neighbor of vertex 3. Similarly, we assign each neighbor for all the vertices and if we are unable to find a neighbor, the default value is set to 0 for the neighbor list.

Part II:

In this part, we are provided with a starting triangle, and a point on the convex hull (outline) of the mesh. We need to find a path through the triangles to the point from the starting triangle. We need to calculate the barycentric co-ordinates of the vertex to find w.r.t. the current triangle to look for the next closest triangle to the point.

Barycentric co-ordinate of a vertex r, is defined by (u, v, w), such that,

r = u **X1** + v **X2** + w **X3** ; and,
u + v + w = 1;

Thus, we form the matrix **T**, such that,

T . *λ* = r

where T is the 3 x 3 matrix formed by the co-ordinates of X1, X2 and X3, the vertex of the triangle, λ is [u, v, w]’ and r is the vertex to be found.

Hence using Cramer’s rule, we find u, v and w for a given triangle, findPoint pair. Now, if the point lies inside the triangle, the values of barycentric co-ordinates are in range (0,1). For a point outside the triangle, the values are negative. If a point is the vertex of the triangle, its barycentric coordinate is 1 and the other two are 0, which is how we find that the triangle in which the search vertex lies is found. Otherwise, if the one of the barycentric coordinate is 0 and the other two are positive and not equal to 1, then the point lies on the line defined by the other two vertex.

Thus, according to the values of barycentric co-ordinates we move to the next closest triangle. The value which is most negative (least among u,v and w) is taken to be the direction in which we have to move. Thus, we use this algorithm iteratively to reach the point being searched.

We have color coded the start triangle to be “Green”, the path followed as “Blue”, i.e the triangles visited on the way are Blue and the final triangle where the point lies is “Red”. If the starting triangle has the vertex to be found, we see only that triangle colored as Red, indicating it is the start and final triangle. If the number of triangle traversed is 2, we see a green triangle which is the start triangle and a red triangle which is the final triangle in order to reach findPoint.

We may note, that taking the most negative value as the indication of the direction in which we need to move, does not always gives the shortest path to the findPoint, although it is mostly the case for a mesh with large number of points. For the most optimum solution we need to take a Dynamic Programming approach and pursue each path corresponding to each negative barycentric coordinate, to find the shortest path to the point from the start triangle.

Part III:

In the third and last part, we had to map the 2D points generated to a primitive function in 3D and generate a corresponding discrete 3D surface which has “numpts” discrete vertices on it. The objective was to study the calculation of gaussian curvatures for such discrete surfaces. In the last homework we learned how to calculate the gaussian curvatures of a continuous surface. The gaussian surfaces for some continuous surface is shown.

We mapped the 2D points generated to a 3D surfaces by evaluating the z-coordinates for the surface on the basis of the x and y coordinates. Thus we obtain three kind of surfaces from the same set of points. To evaluate the desired kind of surface remove the commented line corresponding to the evaluating of z-coordinate. The 3D surface of that shape is generated.

Now we calculate the discrete gaussian curvatures using the formulas discussed in class. An improvement to the discrete curvature KD was proposed in the paper Borrelli et al. which has been implemented as a second discrete gaussian curvature, KD2.

We observe that the calculation of KD and KD2 produce very different color maps for the surfaces for some surfaces while similar color map to some surfaces. This maybe attributed to the fact that the vertex over which curvatures are calculated need not be a boundary vertex since then the vertex calculation gives arbitrarily high or low values skewing the distribution. This can be verified by looking at the KD and KD2 values. Thus, on ignoring the values for discrete gaussian curvature for corner vertices, we obtain a uniform color map over the surface which mimic the gaussian curvature for similar continuous surfaces.

The color map is obtained by normalizing the KD and KD2 values to a [0,1] scale and using these value to plot a color range for each vertex. Thus, each vertex is associated with a color based on its discrete gaussian curvature values for both KD and KD2.