Computational Geometry Lab:
SEARCHING A TETRAHEDRAL MESH

John Burkardt
Information Technology Department
Virginia Tech
http://people.sc.fsu.edu/~jburkardt/presentations/cg_lab_search_tet_mesh.pdf

August 28, 2018

1 Introduction

This lab continues the topic of *Computational Geometry*. Having studied tetrahedrons and tetrahedral meshes (or "tet meshes"), we will now turn to the question of *searching*. In particular, we will suppose that we have a computational region $\mathcal{R}$, and a set of points **P** which are contained in $\mathcal{R}$. (Actually, we will also allow the possibility that some points are outside the region.)

We further suppose that we have constructed a tet mesh τ of $\mathcal{R}$, consisting of **t_num** tetrahedrons, with a typical tetrahedron indicated by $\mathbf{T_i}$. We now wish to know, for each point $\mathbf{p_i}$, the identity of the tetrahedron $\mathbf{T_i}$ that contains the point. If the point is shared by several tetrahedrons, it is sufficient to give just one tetrahedron. If the point does not lie in any of the tetrahedrons, then we may be satisfied with a flag value of "-1", or perhaps we would be interested in the index of a tetrahedron that is near to the point.

We already know, from a previous lab, how to determine whether a point lies inside a tetrahedron. So if we have a tet mesh, and ask which tetrahedron contains a point, we could simply check each tetrahedron, stopping as soon as we find one. This would mean that on average we'd have to check about **t_num**/2 tetrahedrons.

It's hard to imagine, but it's possible to carry out this task much faster. There is a method called the *Delaunay search*, which makes roughly only $\sqrt[3]{\textbf{t\_num}}/2$ checks to locate a point in a tet mesh. It is not unusual for a tet mesh to have a million tetrahedrons, in which case the Delaunay method would be about 10,000 times faster than the straightforward searching method.

This faster method only works on special tet meshes, namely, those that have the *Delaunay property*, which is explained and discussed in another lab. Moreover, a certain amount of preprocessing must be done in order to record which tetrahedrons are neighbors of each other. However, once that is done, the algorithm works beautifully, starting with a random tetrahedron as a first guess, and then "walking" from one tetrahedron to the next, more or less directly towards the correct one.

1.1 Where is a Point Relative to a Tetrahedron?

Let us recall information from the lab on barycentric coordinates for tetrahedrons. We suppose we have a tetrahedron **T**={**a,b,c,d**} and a point **p**. The following function **V(T)** returns a quantity which is equal to the volume of the tetrahedron, possibly multiplied by -1.

$$V = \frac{1}{6} \begin{vmatrix} a.x & a.y & a.z & 1 \\ b.x & b.y & b.z & 1 \\ c.x & c.y & c.z & 1 \\ d.x & d.y & d.z & 1 \end{vmatrix}$$

The fact that there's a possibility of a minus sign isn't so important. The important thing is that the value of $\mathbf{V}(\mathbf{T})$ changes sign in a consistent way. In particular, if we fix three vertices of $\mathbf{T}$ but allow one vertex to move, then $\mathbf{V}(\mathbf{T})$ will stay the same sign until the vertex touches the plane of the other three vertices, and will switch sign if the vertex moves through that plane to the other side.

This property allows us to compute barycentric coordinates of a point $\mathbf{p}$ with respect to a tetrahedron in essentially the same way we did for triangles. The barycentric coordinates of $\mathbf{p}$ are defined as:

$$\xi_a = \frac{\mathrm{V}(p, b, c, d)}{\mathrm{V}(a, b, c, d)}$$

$$\xi_b = \frac{\mathrm{V}(a, p, c, d)}{\mathrm{V}(a, b, c, d)}$$

$$\xi_c = \frac{\mathrm{V}(a, b, p, d)}{\mathrm{V}(a, b, c, d)}$$

$$\xi_d = \frac{\mathrm{V}(a, b, c, p)}{\mathrm{V}(a, b, c, d)}$$

It should be clear that the sum of the barycentric coordinates is 1. Moreover, the values of the coordinates tell us about the position of $\mathbf{p}$. If $\mathbf{p}$ is

- **inside** the tetrahedron, then all the coordinates are positive;

- **on a face**, one coordinate is zero;

- **on an edge**, two coordinates are 0;

- **a vertex**, three coordinates are 0, and one is 1;

- **outside** the tetrahedron, then one, two or perhaps three of the coordinates will be negative.

In particular, if ξ_a is negative, then relative to the face $\{\mathbf{b,c,d}\}$, the point $\mathbf{p}$ lies on one side and vertex $\mathbf{a}$ on the other. Similar statements hold for the other three barycentric coordinates.

1.2 Program #1: Where is a Point Relative to a Tetrahedron?

Write a program which

- reads the definition of a tetrahedron $\mathbf{T}=\{\mathbf{a,b,c,d}\}$;

- reads a point $\mathbf{p}$;

- computes the barycentric coordinates $(\xi_a, \xi_b, \xi_c, \xi_d)$ of $\mathbf{p}$ with respect to $\mathbf{T}$;

- prints $\mathbf{p}$, the barycentric coordinates, and the word "inside" or "outside", depending on whether $\mathbf{p}$ is inside $\mathbf{T}$ or not.

Use the tetrahedron **Tet1** defined by

```
{ {1.0,   2.0,   3.0}
  {4.0,   1.0,   2.0}
  {2.0,   4.0,   4.0}
  {3.0,   2.0,   5.0} }
```

For $\mathbf{p}$, use the points

```
{2.70, 2.40, 3.90},
{2.50, 2.25, 3.50},
{1.15, 3.80, 6.05},
{3.00, 2.00, 5.00},
{2.10, 3.00, 4.10},
{2.60, 3.10, 3.40}
```
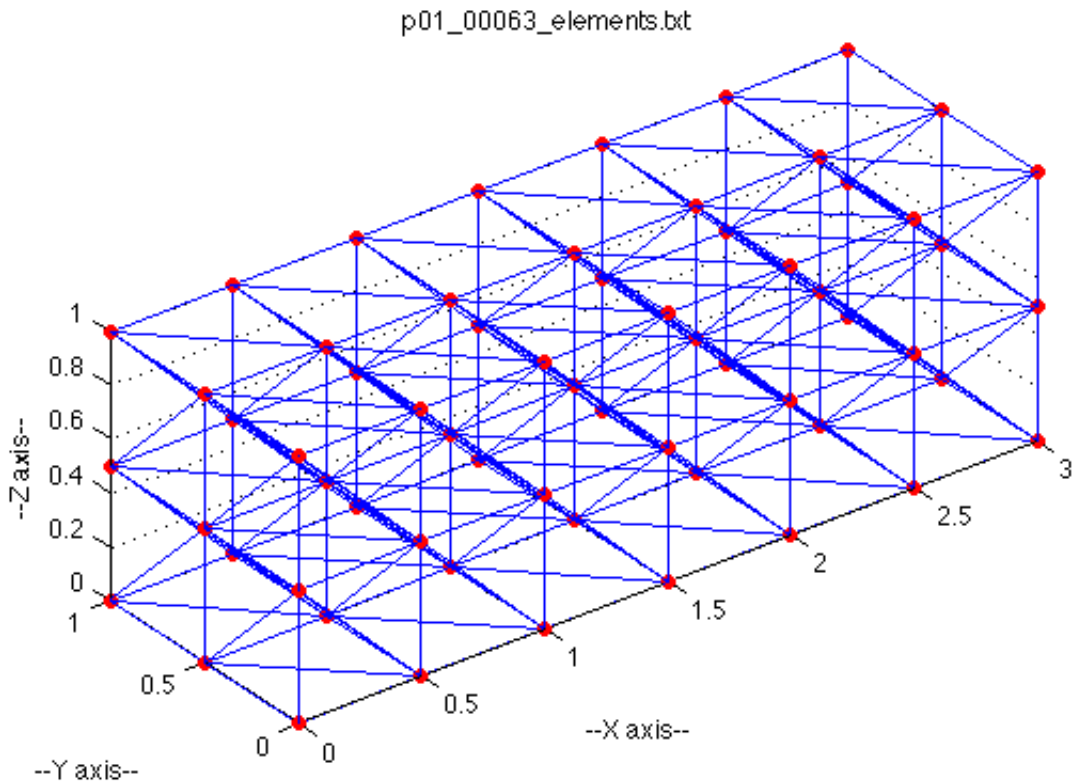
p01_00063_elements.txt

Figure 1: The **Channel** tet mesh.

1.3 Channel, an Example Tet Mesh

For some of the coming exercises, we will want to have a simple tet mesh to work with. We will use the **Channel** tet mesh, which is illustrated and partially defined here. The full definitions of the arrays are available on the web site.

There are 63 nodes in the mesh. The full data is available at "http://people.sc.fsu.edu/∼burkardt/latex/cg_lab_search_tet_mesh/channel_nodes.txt". The first 20 entries of the node coordinate array for **Channel** are:

```
0.0   0.0   0.0
0.0   0.0   0.5
0.0   0.0   1.0
0.0   0.5   0.0
0.0   0.5   0.5
0.0   0.5   1.0
0.0   1.0   0.0
0.0   1.0   0.5
0.0   1.0   1.0
0.5   0.0   0.0
0.5   0.0   0.5
```

```
0.5  0.0  1.0
0.5  0.5  0.0
0.5  0.5  0.5
0.5  0.5  1.0
0.5  1.0  0.0
0.5  1.0  0.5
0.5  1.0  1.0
1.0  0.0  0.0
1.0  0.0  0.5
```

There are 144 elements in the mesh. The full data is available at
"http://people.sc.fsu.edu/~burkardt/latex/cg_lab_search_tet_mesh/channel_elements.txt". The first 20 entries of the node element array are:

```
 1    2    4   10
 2    4    5   10
 2    5   10   11
 2    3    5   11
 4    5   10   13
 3    5    6   11
 5   10   11   13
 4    5    7   13
 5    6    8   14
 5    7    8   13
 6    8    9   14
11   13   14   19
12   14   15   20
 3    6   11   12
 5    6   11   14
 6    9   14   15
 6   11   12   14
 6   12   14   15
 7    8   13   16
 5    8   13   14
```

There are 144 rows in the neighbor array (to be defined shortly). The full data is available at
"http://people.sc.fsu.edu/~burkardt/latex/cg_lab_search_tet_mesh/channel_neighbors.txt". The first 20 sets of neighbors are:

```
 2   -1   -1   -1
 5    3    1   -1
 7   -1    4    2
 6    3   -1   -1
 7   -1    8    2
15   14    4   -1
21   24    5    3
10   -1    5   -1
11   20   15   -1
19   20    8   -1
22   16    9   -1
34   35   21   24
36   42   23   18
17   -1   -1    6
```

```
17   24    9    6
26   18   -1   11
23   18   15   14
13   16   -1   17
25   -1   -1   10
25   24    9   10
```

1.4 Where is a Point Relative to a Tet Mesh?

Now suppose we have a tet mesh τ and a point **p**. How do we determine the index of the tetrahedron **Ti** that contains the point? How quickly can we do this?

The obvious approach, known as the *naive search*, simply checks each tetrahedron. Such a search can take up to **t_num** steps. If the point is guaranteed to be in at least one of the tetrahedrons, and if we stop as soon as we find that tetrahedron, the average number of steps is actually **t_num**/2.

When we "check" a tetrahedron **Ti** to see if it contains **p**, we actually simply compute the barycentric coordinates of **p** with respect to **Ti**. If all coordinates are nonnegative, we have found our tetrahedron. Otherwise, we prepare to check the next tetrahedron in the list.

1.5 Program #2: Where is a Point Relative to a Tet Mesh?

Write a program which

- Reads the element node and node coordinate arrays of a tet mesh τ;

- Reads a point **p**.

- Performs the naive search algorithm to find the tetrahedron containing **p**;

- Prints the tetrahedron index, and the number of steps required.

For your tet mesh, use the **Channel** example. For your point **p**, try each of the following: {0.9,0.9,0.9}, {0.6,0.3,2.4}, {0.4,1.0,2.5}, {0.0,1.0,3.0}, {2.0, 0.5, 1.5}.

It's possible that a point is not contained in *any* tetrahedron. In that case, let's print out the value "-1" in place of the tetrahedron index.

1.6 The Neighbor Tetrahedron Array

In order to apply a faster method for finding points in a tet mesh, we must define the *neighbor tetrahedron array*, a crucial piece of information that will help us find our way to the answer.

The neighbor tetrahedron array is easily defined. If we think about the faces in a tet mesh, most of them are shared by two tetrahedrons. The exceptions occur if a face lies on the exterior boundary, or an internal hole of the region. So, given any tetrahedron in the tet mesh, we can ask, for each of its faces, whether there is another tetrahedron that shares that face, and if so what the index of that tetrahedron is. It will be convenient to call this array **neighbor**, and to assume it has the shape 4 x **t_num**. We will also order the neighbors of tetrahedron **t** so that **neighbor(1,t)** is the neighbor across the face **{b,c,d}**, (and hence opposite the first node, **a**). Similarly, neighbors 2, 3 and 4 share faces **{a,c,d}**, **{a,b,d}** and **{a,b,c}** respectively. It is of course possible that there is *no* neighbor that shares a given face. In that case, we can give that array entry the special value of -1.

Consider the **Channel** tet mesh shown in the figure. Since a 3D figure is hard to label and to view, you'll probably want to refer to the element node table. If the nodes of element 9 are listed in the order {5, 6, 8, 14}, then the 9th row of the neighbor array will be {11, 20, 15, -1}. You can check that tetrahedron 9 shares nodes 6, 8 and 14 with its neighbor tetrahedron 11, which is why the first element of the 9th row of

the neighbors array is 11. Similarly, depending on the ordering of the nodes in the 11th element, one of the entries in the 11th row of the neighbors array must be 9.

These relationships are easy to track down one by one. But is it possible to compute, automatically, the whole array of neighbors, having only the element node array to work with?

Imagine that we needed to know the neighbors of element 18. Tetrahedron 18 has the faces {6,12,14,15}, so it has faces {12,14,15}, {6,14,15}, {6,12,15} and {6,12,14}. To find the neighbors of tetrahedron 18, we have only to search the element node array for the occurrence of these faces. Of course, when looking for a neighbor along the face {12,14,15}, we have to consider the possibility that the nodes may occcur in this order or the reverse order. Then we also have to realize that, because of "wrap around", the nodes 12, 14 and 15 can show up in the element node array in any one of eight ways:

```
{ 12, 14, 15, xx }
{ xx, 12, 14, 15 }
{ 15, xx, 12, 14 }
{ 14, 15, xx, 12 }
{ 12, xx, 15, 14 }
{ 14, 12, xx, 15 }
{ 15, 14, 12, xx }
{ xx, 15, 14, 12 }
```

Since the tetrahedron can have at most one neighbor along a given face, just one of these 8 patterns can occur somewhere in the element node array. We can search for it by looking for the first node 12, then checking to see if the previous or next node is 14, and if so, whether the "next" (going in whichever is the appropriate direction) node is 15. While doing this, we have to think of the four entries in a row of the element node array as wrapping around in a circuit.

The details of programming this search can be painful, but the idea is pretty straightforward. By considering each tetrahedron, and each face of that tetrahedron, and searching the element node array for occurrences of the sequence of nodes that form that face, we can construct the entire neighbor array.

1.7 Program #3: The Neighbor Tetrahedron Array

Write a program which

- Reads the element node array of a tet mesh τ;

- Constructs the tetrahedron neighbor array **neighbor**;

- Prints the tetrahedron neighbor array.

  For input to your program, use the **Channel** tet mesh.

1.8 Where is a Point Relative to a Delaunay Tet Mesh?

Now let us see whether we can come up with a faster answer to our question about finding the tetrahedron in a tet mesh that contains a given point. It turns out that we can only promise a faster answer if we know that our triangulated region is convex, and that the tet mesh is a Delaunay tet mesh. These matters are discussed in the lab on Delaunay tet mesh; for this discussion, we will assume that these conditions are met. Note, however, that the convexity condition means, in particular, that the tet mesh is guaranteed not to have any internal holes, or indentations in the outer boundary.

The Delaunay search algorithm can be thought of as "walking" through the tet mesh, from one tetrahedron to a neighboring one, until the correct one is found. We start at a random tetrahedron. Our movement strategy rests on two key points:

- the barycentric coordinates tell us which direction we should move;

- the neighbor array tells us what tetrahedron lies in each direction.

Let's make this more clear by imagining that our tet mesh is the **Box3** example, that the point **p** is the centroid of tetrahedron **t16**, and that as our random starting point, we choose tetrahedron **t8**.

Our walk begins by asking whether **p** actually lies in tetrahedron **t8**. We answer this question by computing the barycentric coordinates of **p** relative to **t8**. It turns out that two of the coordinates are negative, corresponding to the faces {3,8} and {8,7}. That means that we are on the wrong side of both of those faces. We may choose to move either north or east. Let's say we choose to go north. The neighbor array tells us that we have now moved to tetrahedron **t9**. Again we compute the barycentric coordinates of **p**. This time, only the coordinate corresponding to edge {8,12} is negative, so our only choice is to move east. The neighbor array tells us we've moved to tetrahedron **t10**. We continue in this way until we reach **t8**, when the barycentric coordinates of **p** are all positive, so that we know we've reached our goal.

If we ever try to move in a particular direction, and discover that the neighbor array value is -1, then our search has terminated, and the point **p** actually lies outside the tet mesh. That is only true because we are assuming the tet mesh is convex.

The walking procedure described here can fail if the tet mesh does not have the Delaunay property. In that case, the walk may end up going in a cycle instead of reaching the goal.

To estimate the efficiency of the algorithm, we can start by supposing that we have **t_num** tetrahedrons, that a typical tet mesh has a "width" and "height" and "depth" of about $\sqrt[3]{t\_num}$, and a random starting tetrahedron might therefore be on average about $\sqrt[3]{t\_num}/2$ tetrahedrons distant from the corrrect one. The naive search will typically search **t_num**/2 tetrahedrons, and this means there's a considerable difference in performance between these two search methods.

1.9 Program #4: Where is a Point Relative to a Delaunay Tet Mesh?

Write a program which

- Reads the element node and node coordinate arrays of a tet mesh τ;

- Constructs the tetrahedron neighbor array **neighbor**;

- Reads a point **p**.

- Performs the Delaunay search algorithm to find the tetrahedron containing **p**;

- Prints the number of steps required.

For your tet mesh, use the **Channel** example. For your point **p**, try each of the following: {0.9,0.9,0.9}, {0.6,0.3,2.4}, {0.4,1.0,2.5}, {0.0,1.0,3.0}, {2.0, 0.5, 1.5}.