Computational Geometry Lab: TRIANGULATIONS

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

August 28, 2018

1 Introduction

This lab continues the topic of *Computational Geometry.* Having studied triangles, we will now turn to triangulations, which are collections of triangles. The fundamental role of a triangulation is to analyze a geometric shape or region, making it easy to study by breaking it down into simple shapes. Regions with holes or separate parts can be treated this way, and regions with curved edges can at least be well approximated. With some simple adjustments, triangulations can also be used to define non-planar surfaces, such as the wing of an aircraft, the shape of a teapot, or the surface of the earth.

In this lab, we will not be concerned with how a triangulation is created. That is the topic of a different lab. Here we will assume that a triangulation has been created for us, and we will consider some very basic questions, including how we can represent the triangulation in a computer program. There is much information hidden in the raw representation of a triangle, and we will investigate how the basic data can be analyzed to produce a list of the boundary edges of the triangulation, and an table of the pairs of neighboring triangles that share an edge.

After we have gotten an idea of how to represent a triangulation and work with its data structure, we will be ready to consider a wide variety of tasks of computational geometry. We can find the area of the triangulated region, determine if a point is inside the region and which triangle contains it, we can list the triangles that lie along the boundary, we can compute the integral of a function over the region. We have also laid the groundwork for the finite element method, which will allow us to approximate and solve partial differential equations on the region.

2 A Logical Representation

Our informal definition of a triangulation will simply be that it is a collection of triangles. We'll assume that these triangles don't actual overlap, although they can touch. We'll also assume that any two triangles that touch have exactly one vertex in common, or two vertices and the entire edge between them. This rules out some irregular cases that we don't want to deal with.

Just as a triangle **t** can be thought of as a list of vertices, $t = \{a, b, c\}$, we can think of a triangulation τ as being a list of triangles $\tau = \{t_1, t_2, \ldots, t_n\}$. Of course, this is just a logical representation. We will need to come up with a way to store this information on a computer, in such a way that we can conveniently extract information that we need.

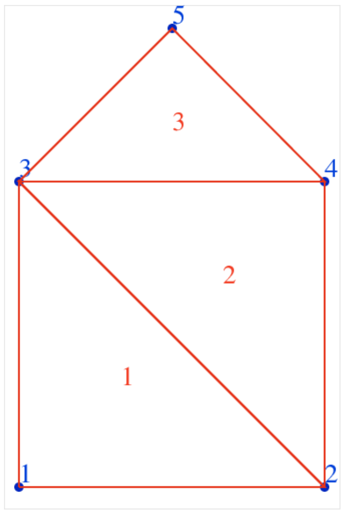

Figure 1: The ”house3” triangulation.

3 A Computational Representation

Because we are dealing with triangulations, we're going to have a lot of triangles, and a lot of vertices. We need a simple format that will allow us to represent the triangulation, either as data in files, or as variables in a program.

The two basic quantities in the triangulation are the nodes and the triangles.

Nodes are simple to represent, since they can be identified simply by their Cartesian coordinates. Thus, to represent the node information, we might need a counter variable **node_num** and a real array **node_xy** of order 2 by **node_num** to contain the coordinates.

To describe the triangles, we first need a counter, which we might call **triangle_num**. Now how do we store the triangles?

The triangles can be defined by listing the nodes that are their vertices. These vertices should *not* be listed by their coordinate values, but rather by their indices in the **node_xy** array. Let us call the array that defines the triangles **triangle_node**. It will be an integer array, of order 3 by **triangle_num**.

(This brings up a side issue of whether we count from 0 or from 1. In these notes, we will generally use indices that start at 1.)

A second issue for triangles is the question of **orientation**. Recall from our lab on triangles that it is useful to consistently describe every triangle with the same counterclockwise orientation ("CCW"). In the "house3" triangulation, triangle number 2 has vertices 2, 4 and 3, listed in CCW order. Thus, we might describe this triangle as any of {2,4,3} or {4,3,2} or {3,2,4}, since all three lists travel along the triangle in the CCW direction. However, it would be inconsistent to describe the triangle using the list {2,3,4}, since this list goes clockwise instead.

To summarize, we now have a simple scheme for representing a triangulation involving four variables:

- **node_num**, the number of points;
- **node_xy**, a 2 by **node_num** real array of points;
- **triangle_num**, the number of triangles;
- **triangle_node**, a 3 by **triangle_num** integer array of node indices, listed in CCW order.

For instance, here is the data structure associated with the triangulation of a child's drawing of a house:

```
  node_num = 5;

  node_xy = {  { 0.0, 0.0 },
               { 2.0, 0.0 },
               { 0.0, 2.0 },
               { 2.0, 2.0 },
               { 1.0, 3.0 } };

  triangle_num = 3;
  triangle_node = {  { 1, 2, 3 },
                     { 2, 4, 3 },
                     { 3, 4, 5 } };
```

If you think about our definition of a triangulation, we're loosely suggesting that it's a list of triangles, while each triangle is a list of (three) points, and each point is a list of two coordinates.

Small triangulations are easy to store internally. Bigger triangulations may need to be read from files, one containing the point coordinates, and one the point indices that define the triangles.

We will follow a simple convention in naming the two files that define a triangulation. If we refer to a triangulation as the "elbow3" data, then that means that the two files will be called "elbow3_nodes.txt" and "elbow3_triangles.txt". A number of such files are available at
http://people.sc.fsu.edu/~burkardt/data/triangulation_order3/triangulation_order3.html.

4 Program #1: Read and Store a Triangulation

Our first program will simply read a pair of files defining a triangulation and print out the data.

The ideal program would be invoked by a command like this:

```
  tri_read house3
```

It would be able to read the string "house3" from the command line, figure out the names of the two triangulation files, read the data inside them, and print that data.

Things you may not know how to do might include:

- how a program can read a command line argument, so that the **tri_read** program knows that it's supposed to read the "house3" data
- how to read an input string from the user and append "_nodes.txt" to it to make the name of the node file
- how to open text files for reading
- how to read an unknown amount of data into a program

All of these things are worth learning, but if you have an unreasonable amount of difficulty in getting any one of them to work, please ask for help!

Using your program, read in the "house3" triangulation data.

The node file is called "house3_nodes.txt" and looks like this:

```
0.0  0.0
2.0  0.0
0.0  2.0
2.0  2.0
1.0  3.0
```

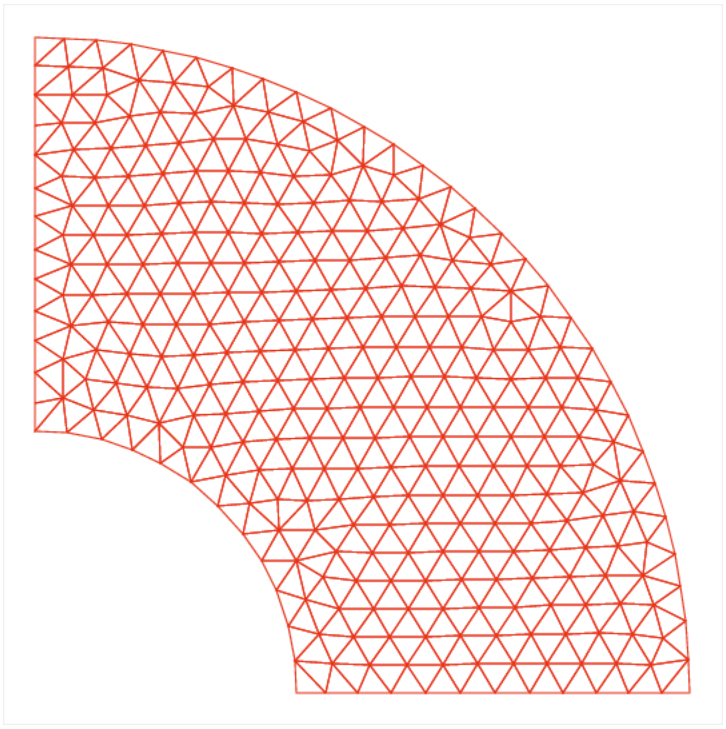
Figure 2: The “elbow3” triangulation.

The triangle file is called “house3_triangles.txt” and looks like this:

```
1  2  3
2  4  3
3  4  5
```

Your program should print out the following information:

- the name of the triangulation data: “house3” in this case;
- the number of nodes, and the number of triangles;
- the coordinates of each node;
- the indexes of the nodes that form each triangle.

If you are satisfied with your program’s performance on “house3”, try the “elbow3” dataset as well.

5 The Area of a Triangulation

An important property of a triangulation is its area. Since a triangulation is made up of triangles, and we require that these triangles not overlap, the area of the triangulation is the sum of the areas of the triangles. From our previous lab on the properties of triangles, we know how to compute the area of one triangle.

In order to compute the area of a triangulation, we must understand how to write a program that reads in a triangulation, processes one triangle at a time, and for that triangle, correctly determines the vertex coordinate information needed to compute the area. This involves using the indices in **triangle_node** to access coordinates in the **node_xy** array.

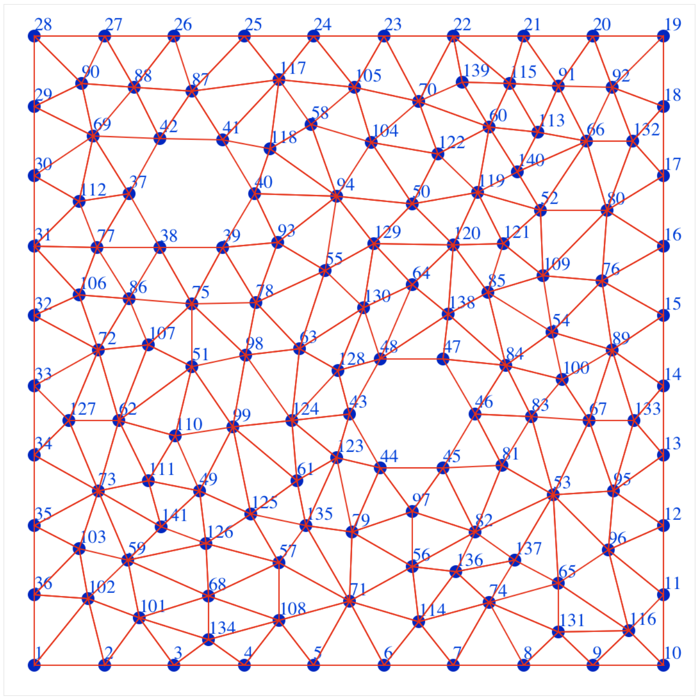

Figure 3: The “hex_holes3” triangulation.

6 Program #2: The Area of a Triangulation

Write a program that reads a triangulation and computes and prints its area.

Debug your program using the “housed” triangulation, whose area is 5.0 square units, Now test your program by computing the areas of the “elbow3” and “hex_holes3” triangulations.

7 The Boundary of a Triangulation

Our simple data structure stores a triangulation, but there are a number of interesting facts about that triangulation, hidden within the data, which we may need to know. We will start by considering the very simple question of the *boundary* of the triangulated region. Since the boundary is formed by some of the edges of the triangles, a crude description of the boundary would be a list of those edges. Thus, for the “house3” triangulation, the boundary would be

```
house3.boundary = { {1,2},{2,3},{4,5},{5,3},{3,1} }
```

Notice that in this list we have given each edge with the same orientation that it has as part of its triangle. We have also listed the edges in a way which happens to form a closed curve, that is, the second node of each edge is the first node of the next edge. Giving the boundary this way makes it easier to carry out other tasks.

Looking at a picture of a triangulation, it’s easy to see the boundary. But staring at the data in our two files, it’s not so clear what is going on. However, there is a fairly easy way to understand how the boundary can be defined. The key to this idea is related to our insistence that our triangles all be oriented in the same CCW sense.

So let’s imagine drawing the triangulation by drawing every triangle. That’s actually a bit wasteful, because it means every line will be drawn twice, since it’s shared by two triangles. But wait a minute, there will be some lines drawn only once . . . because they are only part of one triangle . . . and those lines are exactly the lines that form the boundary.

There's even more going on with this idea. Suppose that our triangulation includes an interior hole. The lines along this interior hole will also only be drawn once, so it sounds like they should be included as part of the boundary. Perhaps, though, we should distinguish this boundary by calling it an *internal boundary* as opposed to the *external boundary*.

Interestingly, if we follow the external boundary along the orientation of the lines that make it up, we go in the CCW orientation. In contrast, the internal boundary will be traced in the CW orientation! This is a useful distinction to keep in mind, and it makes sense geometrically.

If we have multiple boundaries like this in a triangulation, then it is useful for other computations if we are careful to list the boundary components one at a time, and for each component, give the edges in order as suggested earlier in the "house3" discussion.

8 Algorithm: Finding the Boundary Edges by Brute Force

It may be hard to see how we can compute the boundary of an arbitrary triangulation. In order to get started, let's just use a brute force approach. It won't be clever, but it will get the job done.

Our algorithm is guided by the idea that the boundary is the set of all edges that only occur once. From the **triangle_node** array we can make a list of edges, and then cross off the duplicates (remembering that the second occurrence of an edge will show up with the opposite ordering of the nodes). Here's the edge list for "house3", with the two pairs of duplicates already marked:

```
1 2
2 3<---+
3 1    |
2 4    |
4 3<---|-----+
3 2<---+     |
3 4<---------+
4 5
5 3
```

Now you can see that {2,3} and {3,2} cancel out, and so do {4,3} and {3,4}. Everything else is a boundary edge, so we've got our list!

Once we have gotten the boundary edges, we should try to order the edges. Assuming the boundary involves only a single component, we can pick any edge, say {1,2} to start with. Then we try to match the second node of this edge against the first node of a new edge. This leads us to the edge {2,4}, and we continue until we finish with edge {3,1}.

9 Program #3: Finding the Boundary Edges

Write a program that reads in a triangulation, computes the list of boundary edges, arranges the boundary edges into a connected list, and prints the list out.

Debug your program on the "house3" triangulation. Then try it on the "elbow3" triangulation.

10 Doing a Better Job

Our brute force boundary calculation was simple, and not difficult to program. However, it was not efficient. That is, if we let **N** be the number of edges in the triangulation, the process of comparing each edge with all the other edges in the list requires on the order of N^2 operations.

There are more efficient approaches. In particular, we note that setting up a sorted list of **N** objects takes about $N \log N$ operations. So here we sketch another method of computing the boundary edges that will run more quickly.

Instead of immediately creating a list of edges, we imagine instead that we start with an empty list. Whenever we add or remove items from this list, we will do so in a way that keeps it sorted in order. We begin by examining the edges of the first triangle. If an edge of this triangle appears in the list, then we remove that occurrence from the list. Otherwise, the triangle's edge is added to the list (inserted in the correct spot that preserves the sorted ordering.) Because the list is always in sorted, our searches for a match occur much more quickly than with the brute force search.

The extra complications in programming this procedure are rewarded by a much improved running time. We will not require you to investigate this topic further. However, it's important to be aware of the fact that the straightforward solution to a problem can often be greatly improved with a little thought.

11 Multiple Boundaries

The second stage of the boundary program strings the edges together to make a connected curve that traces the boundary of the region. What happens if the region has holes? In that case, the boundary will not be a single connected curve, but several. So if we string together the boundary edges, we will return to our starting node but will have unused edges left over. That means we need to keep track of how many boundary edges we found, and how many we have strung together. As long as there are boundary edges remaining, we have at least one more boundary curve to follow. So we can pick any unused edge, and start tracing that curve til we return to our start, repeating this process for as many holes as we have to deal with.

12 Program #4: Multiple Boundaries

Modify your boundary program so that it can handle the case of multiple boundary curves.

Use "hex_holes3" to test your program. By looking at the diagram of this region, what edges do you expect to see? In what orientation will these edges come out?

Can your program figure out which of the boundary curves is the one that corresponds to the outer boundary, and which ones are holes in the region?

13 Neighboring Triangles

We say that two triangles t_1 and t_2 in a triangulation are *neighbors* if they have a common edge. We assume that our triangles have a common CCW orientation. If we suppose that in triangle t_1, the common edge involves nodes **p** and **q** in that order, then in t_2 the nodes must be listed in the opposite order.

It can be useful to make a list of the neighbor triangles of each triangle. We will call this list the **triangle_neighbor** array. We can think of it as a table of **triangle_num** rows and 3 columns.

Entry **triangle_neighbor(i,j)** will identify triangle t_i's neighbor on the j-th side. In order to define sides, we will say that side #1 of a triangle is the side opposite from vertex #1, and so on. In a few cases, a triangle will have no neighbor on a given side, because that side lies along a boundary. In that case, we can store a special value in the array entry to indicate this. For now, let us simply store the value -1 when there is no neighbor.

For our simple "house3" triangle, our array will be mostly boundary entries:

```
triangle_neighbor = { {  2, -1, -1 },
                      {  3,  1, -1 },
                      { -1, -1,  2 } };
```

Every entry in this array means something, and is precisely where it should be. Make sure you could determine these entries based entirely on the values in **triangle_node**!

It's not too hard to set up the **triangle_neighbor** array if we are allowed to draw a picture. Since the computer has to work from the information in **triangle_node** array, let us think of how we can do this.

Suppose we need to determine the value of **triangle_neighbor(i,j)**. We start out by initializing the entry to -1, in case we don't find a neighbor. We are working with triangle t_i, whose vertices are **a, b, c**. Depending on the value of j, we now need to consider the sides {**b, c**} or {**c,a**} or {**a,b**}. Let's suppose that j is 2, so we are working with side {**c,a**}.

If there is a neighbor of t_i on side j, that neighbor will have an edge of the form {**a,c**} (because the orientation is the opposite). To find whether this neighbor exists, we look through the entire **triangle_node** array. We are searching for any occurrence of the node **a**. If we find it in row k, we then check whether whether the very next node is **c**. (Warning! If **a** occurs in column 3 of **triangle_node**, then the "next" entry is in column 1!) If so, we have found that the neighbor of triangle t_i on side j is triangle t_k, so we set **triangle_neighbor(i,j)=k**.

14 Program #5: Triangle Neighbors

Write a program which reads in a triangulation and computes and prints the **triangle_neighbor** array.

Debug your program using the "house3" triangulation.

Test your program on the "hex_holes3" triangulation. The first 5 rows of your results should be:

```
  -1     4    67
  -1    40   158
 149    84   176
   1   115    62
 193    80   194
```

This program is challenging! Take your time, and be careful.

When you search for neighboring triangles, you will probably simply do a brute force search, as we did for the boundary edges. You may want to think about how it would be possible to use sorting to somehow make this search go faster.