Determining Voronoi Neighbors using MATLAB's VORONOIN Command http://people.sc.fsu.edu/∼jburkardt/presentations/… …voronoi_neighbors.pdf

John Burkardt,
Department of Scientific Computing,
Florida State University, Tallahassee, Florida

September 24, 2016

The Voronoi diagram of a set of points in the plane divides the plane so that each point is assigned a separate polygons, although a few of these polygons will actually be unbounded. Once the plane is subdivided in this way, it is often of interest to determine the list of points which are immediate Voronoi neighbors of a point I. One reason for gathering this information is that the Voronoi diagram may be used in constructing a discrete model of some physical process involving transport, such as heat transfer. In such cases, the quantity being studied will be assigned to each point, and is allowed to flow over time across boundaries with the immediate neighbors. Thus, a computational model will require knowing the neighborhood relations.

There are many ways to compute the Voronoi diagram of a set of points. Within MATLAB, there are two commands, **voronoi()** and **voronoin()**. It turns out that the **voronoin()** command returns enough information to answer our Voronoi neighbor question, so we will now explain how this information is wrapped up in the output, and how it can be unfolded and interpreted.

When MATLAB constructs the Voronoi diagram, it needs a way to indicate that some of the polygons are unbounded. It does this by adding a point at infinity, and pretending that the infinite regions all include that point as a vertex, so from now on, we can pretend that every point is contained in a closed polygon defined by the Voronoi diagram. Two points are *neighbors* if their polygons share an edge. So our question becomes, how do we take the information that MATLAB returns to describe a Voronoi diagram, and analyze it so that we can determine which points are neighbors?

Let's use the following array of 9 points as an example:

```
0.0  0.0
```

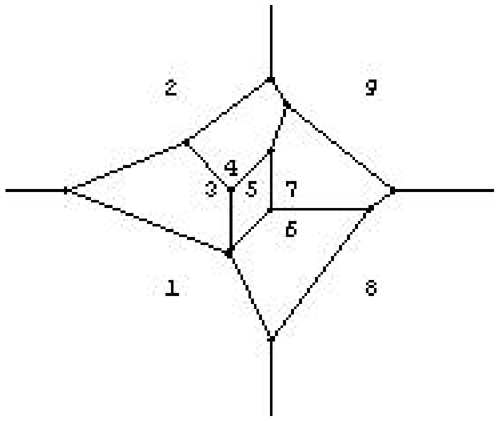

Figure 1: Mathematica's plot of the example data.

```
0.0  1.0
0.2  0.5
0.3  0.6
0.4  0.5
0.6  0.3
0.6  0.5
1.0  0.0
1.0  1.0
```

For this small problem, we could answer our question by computing the diagram with Mathematica's **VoronoiDiagram**[] command and then calling **DiagramPlot**[] to get Figure 1 so that from the illustration, we can easily see that, for example, point 1 has neighbors 2, 3, 6 and 8.

While we can see this picture, the computer can't, and now we need to determine a way in which this same neighbor information can be determined computationally, without the use of eyeballs, for larger problems, and using MATLAB. Let us now consider how to proceed.

If I call **voronoin()** with the example data:

```
[ v, g ] = voronoin ( xy )
```

then the vertices **v** are returned as:

```
    Inf        Inf
  0.5000   -0.2500
 -0.5250    0.5000
  0.2875    0.1750
  0.3000    0.2000
```

```
1.1125    0.5000
0.9875    0.4000
0.5000    0.4000
0.0643    0.7357
0.3000    0.5000
0.5000    1.0625
0.5000    0.7000
0.5763    0.9289
```

But what is more interesting is the entries of the cell array **g**, which contain, for each node, the sequence of vertices that form its boundary. Because **g** is a cell array, we need to handle it a little differently than an ordinary MATLAB array. In particular, we could print the vector of entries that form each row with the commands:

```
for i = 1 : length ( g )
  disp ( g{i} )
end
```

For our data, the **g** information is:

```
1:    4     2     1     3
2:   11     1     3     9
3:   10     5     4     3     9
4:   13    11     9    10    12
5:   12     8     5    10
6:    8     5     4     2     7
7:   13     6     7     8    12
8:    7     2     1     6
9:   13     6     1    11
```

Now, just like two people in the suburbs are neighbors if they share a fence, two nodes in the Voronoi diagram are neighbors if they share an edge. The edges for node 1 are (4,2), (2,1), (1,3) and (3,4). If I rewrite these edges so the pairs of nodes are sorted, and then sort these edges by first element, I have the edges (1,2), (1,3), (2,4) and (3,4). Node 1 is a neighbor of node I if and only if node I also uses one of these edges to bound its polygonal region. So if I list *all* the edges down the left hand side, and all the nodes along the top, I can construct the following diagram, showing which pairs of nodes are connected by an edge:

```
             1    2    3    4    5    6    7    8    9

( 1, 2)      1-----------------------------------8
( 1, 3)      1----2
( 1, 6)                                          8----9
( 1,11)        2----------------------------------9
( 2, 4)   1------------------------6
( 2, 7)                            6---------8
( 3, 4)   1---------3
( 3, 9)        2----3
( 4, 5)             3--------------6
( 5, 8)                      5-----6
( 5, 9)
( 5,10)             3--------5
( 6, 7)                                 7----8
( 6,13)                                 7---------9
( 7, 8)                            6----7
( 8,12)                      5----------7
( 9,10)             3---4
( 9,11)        2--------4
(10,12)                 4---5
(11,13)                 4-------------------------9
(12,13)                 4---------------7
```

From this diagram, we can immediately see that the nodal neighbor array is:

```
     1  2  3  4  5  6  7  8  9
   +--------------------------
 1 | 0  1  1  0  0  1  0  1  0
 2 | 1  0  1  1  0  0  0  0  1
 3 | 1  1  0  1  1  1  0  0  0
 4 | 0  1  1  0  1  0  1  0  1
 5 | 0  0  1  1  0  1  1  0  0
 6 | 1  0  1  0  1  0  1  1  0
 7 | 0  0  0  1  1  1  0  1  1
 8 | 1  0  0  0  0  1  1  0  1
 9 | 0  1  0  1  0  0  1  1  0
```

Thus, we see that **voronoin()** returns in the second output argument **g** enough information to determine which nodes are neighbors.

Now an interesting question is to write a MATLAB M file that can automate this process, that is, accept a set of **n** points in the plane, and return the (sparse) nodal neighbor array, so that **A(i,j)** is 1 if and only if nodes **i** and **j** are Voronoi neighbors.

Talha Arslan, of the Department of Statistics, at Eskisehir Osmangazi University, has suggested the following procedure, which computes the Voronoi adjacency matrix from the information provided by **voronoin()**:

```
[ n, dim ] = size ( x );
```

```
%
%  V contains the Voronoi vertices,
%  C contains indices of Voronoi vertices that form the (finite sides of the)
%  Voronoi cells.
%
  [ V, C ] = voronoin ( x );
%
%  Two nodes are neighbors if they share an edge, that is, two Voronoi
%  vertices.
%
  vn = sparse ( n, n );

  for i = 1 : n
    for j = i + 1 : n
      s = size ( intersect ( C{i}, C{j} ) );
      if ( 1 < s(2) )
        vn(i,j) = 1;
        vn(j,i) = 1;
      end
    end
  end
```

Carrying out this procedure for the test data set, we get:

```
 Voronoi edges:

    4     2     1     3
   11     1     3     9
   10     5     4     3     9
   13    11     9    10    12
   12     8     5    10
    8     5     4     2     7
   13     6     7     8    12
    7     2     1     6
   13     6     1    11

Voronoi adjacency:

Col:   1    2    3    4    5    6    7    8    9
Row
  1:   0    1    1    0    0    1    0    1    0
  2:   1    0    1    1    0    0    0    0    1
  3:   1    1    0    1    1    1    0    0    0
  4:   0    1    1    0    1    0    1    0    1
  5:   0    0    1    1    0    1    1    0    0
  6:   1    0    1    0    1    0    1    1    0
  7:   0     0     0     1     1     1     0     1     1
  8:   1     0     0     0     0     1     1     0     1
  9:   0     1     0     1     0     0     1     1     0
```

Thus, we see that the **voronoin()** command does return all the information we need to establish the Voronoi adjacency matrix, and we also see how we can construct this matrix automatically.