HomeWork 8

1. The matlab code for creating triangular finite element grids :

```
function [ sort, DT_I, P, Val, DT,M ] = generate_grid( h )
figure()
axis([0 2 0 2]);
axis equal;
M = floor(2/h) - 1;
sort = [];
Val = [];
P = [];
for i = 0:h:2
   for j = 0:h:2
      P = [P;j i];
   end
end
total_number = size(P,1);
n1 = 1;
n2 = M+2;
n3 = (M+2)*(M+2);
n4 = (M+2)*(M+1)+floor((M+2)/2)+1;
n5 = (M+2)*floor((M+2)/2)+floor((M+2)/2)+1;
n6 = (M+2)*floor((M+2)/2)+1;
for id = 1:total_number
   if (mod(id,M+2) == 1) && (id <= n6)
      sort = [sort;1];
      Val = [Val;1];
   elseif mod(id,M+2) == 0
      sort = [sort;1];
      Val = [Val;0];
   elseif mod(id,M+2) <= floor((M+2)/2) && mod(id,M+2) >= 1 && id > n5
      sort = [sort;2];
      Val = [Val; NaN];
   else
      sort = [sort;0];
      Val = [Val; NaN];
   end
end
C = [n1 n2;n2 n3;n3 n4;n4 n5;n5 n6;n6 n1];
plot(P(C'),P(C'+size(P,1)),'-r','LineWidth', 3);
DT = delaunayTriangulation(P, C);
IO = isInterior(DT);
hold on
triplot(DT(IO, :),DT.Points(:,1), DT.Points(:,2),'LineWidth', 1)
hold off
DT_I = DT(IO,:);
end
```

Explanation: In the program, the matrix *sort* stores the kinds of points, i.e., 1 represents the known points, 0 represents the unknown points and 2 represents thepoints in the left upper corner outside the L-shape region. The matrix *Val* stores the values of points and the values of unknown points are all *NaN* at first. And the matrix *P* stores the coordinates of all points. The information of these three matrixes corresponds to each other. The triangular finite elements are produced using matlab command *delaunayTriangulation()*. These triangles are produced with uniform size and random direction. Each row in the returned matrix DT could represent an element(triangle) and it stores three values which are the id of points within this element. These ids can help figure out all the local matrixes. The parameter of the function h represents the size of each element.

Result: Figure 1 to Figure 4 are the images of the finite element grids with h = 0.2, h = 0.1, h = 0.05, h = 0.03.

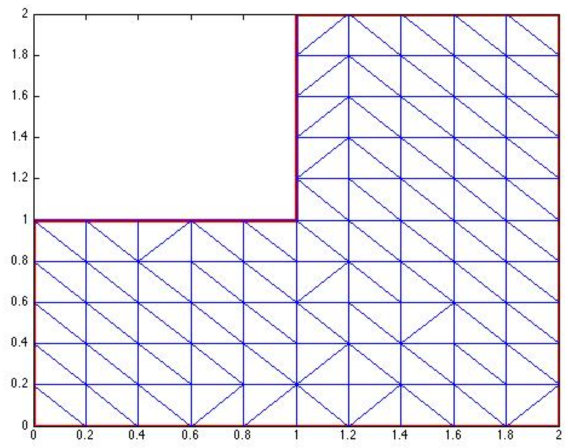


Figure 1. The finite element grids with h = 0.2

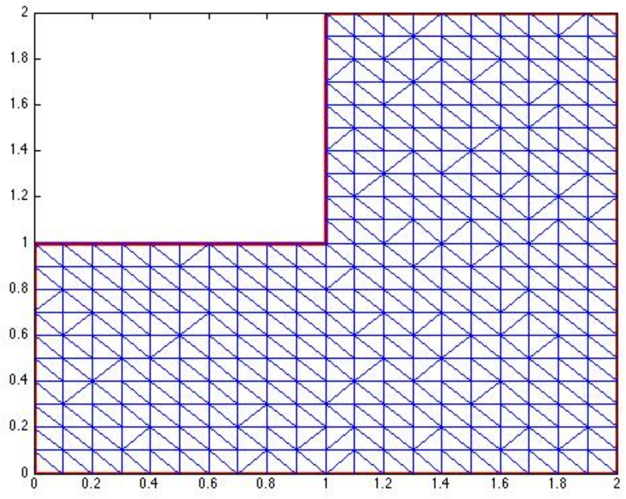


Figure 2. The finite element grids with h = 0.1

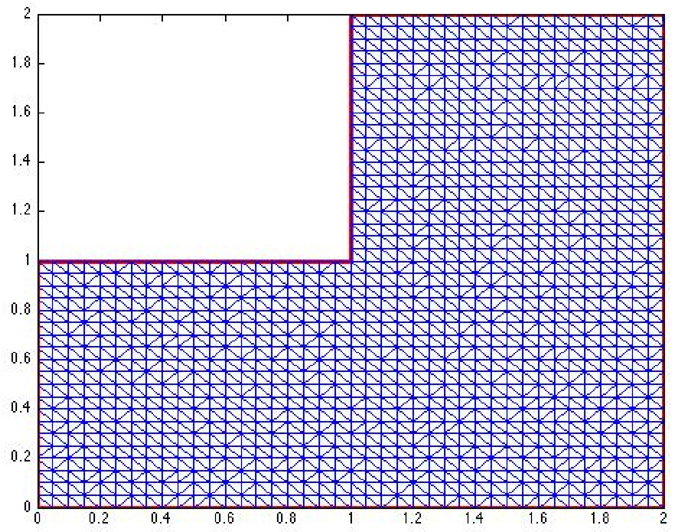


Figure 3. The finite element grids with h = 0.05

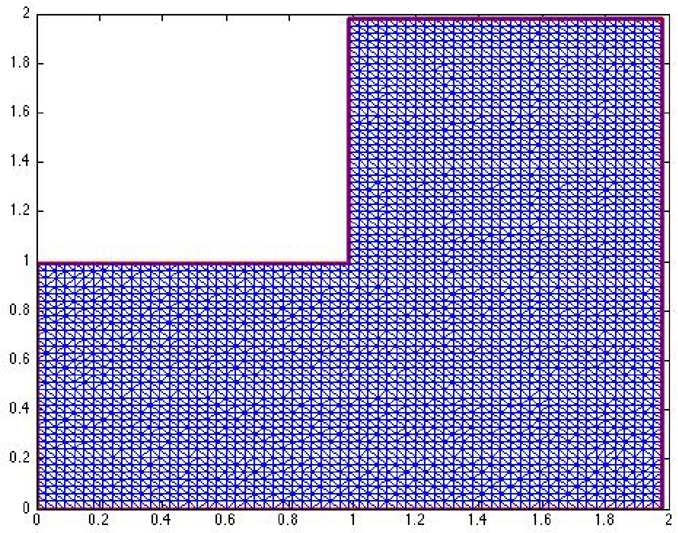


Figure 4. The finite element grids with h = 0.03

This is my first version of function **generate_grid.** However, I have to deal with the points outside the L-region and it's troublesome. Hence, I wrote another version and I only produce points inside the L-region. The code is shown as below,

```
function [ sort, P, Val, DT, DT_I ] = generate_grid( h )
figure()
axis([0 2 0 2]);
axis equal;
M = floor(2/h);
h = 2/M;
sort = [];
```

```
Val = [];

P = [];
for i = 0:h:1
    for j = 0:h:2
        P = [P;j,i];
    end
end
for i = 1+h:h:2
    for j = 1:h:2
        P = [P;j,i];
    end
end
total_number = size(P,1);
for id = 1:total_number
if P(id,1) == 0 && P(id, 2) == 0
        n1 = id;
    elseif P(id,1) == 2 && P(id,2) == 0
        n2 = id;
    elseif P(id,1) == 2 && P(id,2) == 2
        n3 = id;
    elseif P(id,1) == 1 && P(id,2) == 2
        n4 = id;
    elseif P(id,1) == 1 && P(id,2) == 1
        n5 = id;
    elseif P(id,1) == 0 && P(id,2) == 1
        n6 = id;
    end
    if P(id,1) == 0 && P(id,2) <= 1
        sort = [sort;1];
        Val = [Val;1];
    elseif P(id,1) == 2
        sort = [sort;1];
        Val = [Val;0];
    else
        sort = [sort;0];
        Val = [Val;NaN];
    end
end
C = [n1 n2;n2 n3;n3 n4;n4 n5;n5 n6;n6 n1];
plot(P(C'),P(C'+size(P,1)),'-r','LineWidth', 3);
DT = delaunayTriangulation(P, C);
IO = isInterior(DT);
hold on
triplot(DT(IO, :),DT.Points(:,1), DT.Points(:,2),'LineWidth', 1)
hold off
DT_I = DT(IO,:);
end
```

2. The Whole Process:

(1) Using function **Diff_Triangle** to figure out the geometric cofficients a_i, b_i, c_i. The cofficient a_i is used to help figure out the right hand side. Howeveer, in this case, the Lapace's equation is $\nabla^2 u = 0$, hence, f = 0. In addition, the Neumann conditio $\partial u/\partial n = 0$ is automatically satisfied, so the r.h.s of each local matrix are all zero vertors and a_i is not needed.

The matlab code of function Diff_Triangle:.

```
function [b,c,Area_A] = Diff_Triangle( x,y )
Area = x(2)*y(3) - x(3)*y(2) + x(3)*y(1) - x(1)*y(3) + x(1)*y(2) - x(2)*y(1);
Area_A = abs(Area)/2;
b = zeros(3,1);
c = zeros(3,1);
for i = 1:3
  i1 = mod(i,3) + 1;
  i2 = mod(i1,3) + 1;
  b(i) = (y(i1) - y(i2))/Area;
  c(i) = (x(i2) - x(i1))/Area;
end
end
```

EXPLANATION: The returned value *Area_A* is *A* appears in the textbook, *b(i)* is just b_i/2A and *a(i)* is just a_i/2A. The input vectors x and y store the coordinates of the three points.

(2) Figure out the local matrix using the geometrix cofficients. The r.h.s are all zeros.

The matlab code of functin Local_Matrix used to construct local matriexes for each element:

```
function [ A ] = Local_Matrix(x,y)
[b,c,Area_A] = Diff_Triangle(x,y);
A = zeros(3,3);
for i = 1:3
   for j = 1:3
       A(i,j) = Area_A*(b(i)*b(j) + c(i)*c(j));
   end
end
end
```

(3)Combine all the local matrixes into a global matrix, apply the boundary conditions into r.h.s and modify the global matrix. The main idea of the combination process is $a_{g[i]g[j]} \leftarrow a_{g[i]g[j]} + a_{ij}$, where g[i] and g[j] is the id of the three points in the global matrix. And the main idea of dealing with boundary condition is that if u_i is known, then set all elements in the ith row and the ith column 0 and set element a_{ii} 1, in addition, set the ith element in r.h.s equal to u_i and subtract $a_{ji}u_i$ from the jth element in r.h.s unless u_j is known as well. I wrote two versions of the combination process.

The first version:

```
function [ Ag, fg ] = Global_Matrices(vert, Al, Ag, fg)
for row = 1:3
   for col = 1:3
      if vert(col).sort == 1
         if col ~= row
            if(vert(row).sort ~= 1)
            fg(vert(row).id) = fg(vert(row).id) - Al(row, col)*vert(col).value;
            end
            Ag(vert(row).id, vert(col).id) = 0;
         else
            Ag(vert(row).id, vert(col).id) = 1;
         end
            fg(vert(col).id) = vert(col).value;
      elseif vert(row).sort == 1
            Ag(vert(row).id, vert(col).id) = 0;
            fg(vert(row).id) = vert(row).value;
      else
         Ag(vert(row).id,vert(col).id) = Ag(vert(row).id,vert(col).id) + Al(row,col);
      end
   end
end
end
```

EXPLANATION: This function combines the local matrix while modifying the global matrix and r.h.s at the same time. The input consists of Al(the local matrix), fl(the local r.h.s), Ag(the global matrix before the combination), fg(the r.h.s before the combination) and vert which stores the corresponding ids, sorts and values of the three points in the local matrix.

The second version:

```
function [ Ag, fg ] = Global_Matrices1(vert, Al, fl,  Ag, fg)
for i = 1:3
   fg(vert(i).id) = fg(vert(i).id) + fl(i);
   for j = 1:3
      Ag(vert(i).id,vert(j).id) = Ag(vert(i).id, vert(j).id) + Al(i,j);
   end
end
end
```

EXPLANATION: The function **Global_Matrices** finishes the combination process by traversaling all the local matrixes. Then use the codes below to deal with boundary conditions. The matrix D stores the ids of the known points.

```
D = [];
 for i = 1:total_number
      if sort(i,1) == 1
```

```
            fg(:,1) = fg(:,1) - Ag(:,i).*Val(i);
            Ag(i,:) = 0;
            Ag(:,i) = 0;
            Ag(i,i) = 1;
            D = [D;i];
        end
 end
 for i = 1:size(D,1)
         fg(D(i)) = Val(D(i));
 end
```

I wrote the function **get_Aandb** to convert the finite element grids problem into a solving linear system Ax=b problem using the three functions above. The function will return the gloabl matrix Ag and the r.h.s.

```
function [ Ag, fg ] = get_Aandb( DT_I, sort, P, Val )
 element_number = size(DT_I,1);
 total_number = size(P,1);
 M = sqrt(total_number) - 2;
 Ag = zeros(total_number, total_number);
 fg = zeros(total_number,1);
 for i = 1:element_number
     x = zeros(3,1);
     y = zeros(3,1);
     vert(3,1) = struct('id',[],'value',[],'sort',[]);
     for j = 1:3
     x(j) = P(DT_I(i,j),1);
     y(j) = P(DT_I(i,j),2);
     vert(j).id = DT_I(i,j);
     vert(j).sort = sort(vert(j).id);
     vert(j).value = Val(vert(j).id);
     end
     Al = Local_Matrix(x,y);
     [ Ag, fg ] = Global_Matrices(vert, Al, Ag, fg);
 end
 del = [];
 for i = floor((M+2)/2)+1:M+1
     for j = 1:floor((M+2)/2)
         t = i*(M+2) + j;
         del = [del;t];
     end
 end
 Ag(del,:) = [];
 Ag(:,del) = [];
 fg(del) = [];
end
```

Using these codes, solved the unknows in these finite element grids with size h = 0.05, and then visualized the grids with command **trisurf(DT(:,:),DT.Points(:,1), DT.Points(:,2), res).** Here, res is the solution of all the unkowns. I got the visualization as Figure 5 shows.

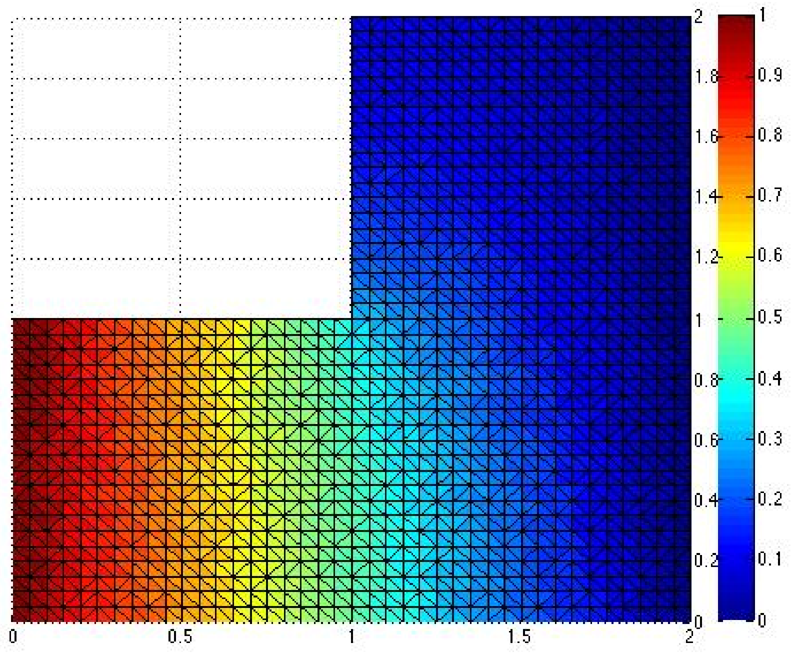


Figure 5. The visualization of finite element grids with h = 0.05

3. Compared the Finite Element Method with the Five-Point Finite Difference Method with different values of h using bicg method.

Table 1. The comparsions of cputimes

CPUTIME	h=0.02	h=0.03	h=0.05
FE	81.8700	11.68	1.3
FD	71.93	9.45	1.04

Obviously, with the same number of elements, the method FE needs more time than the FD method.

The figures below are the iso-value lines for both FE and FD methods with h = 0.02, h = 0.03, h = 0.05. The figures on the left side are the iso-value lines using FD and those on the right side are those using FE.

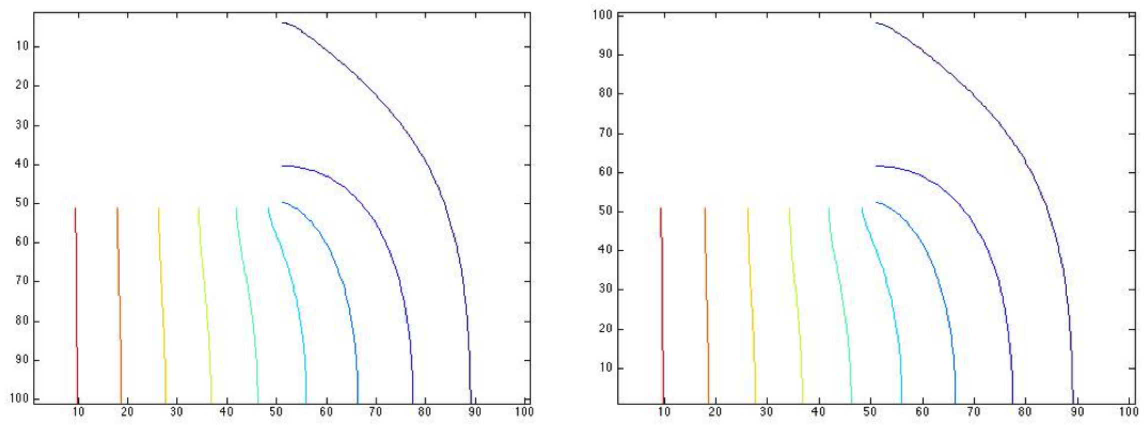


Figure 6. The iso-value lines for FD and FE with h = 0.02

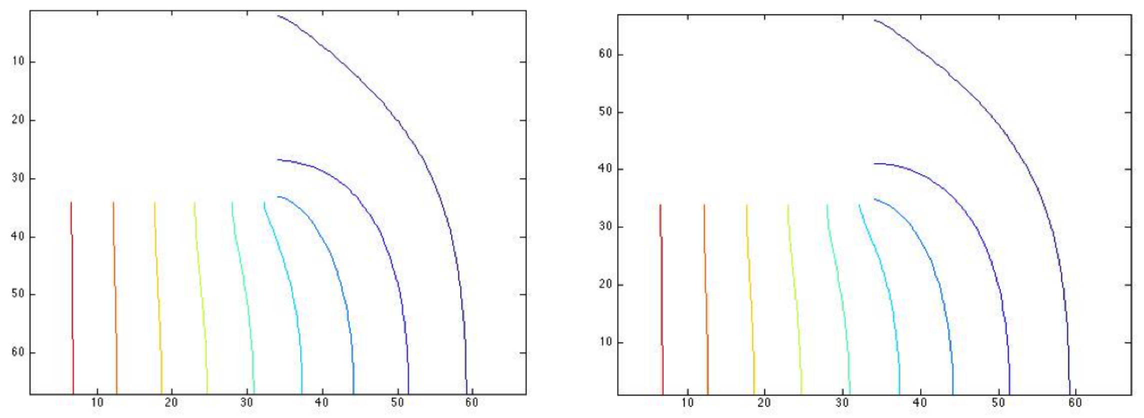


Figure 7. The iso-value lines for FD and FE with h = 0.03

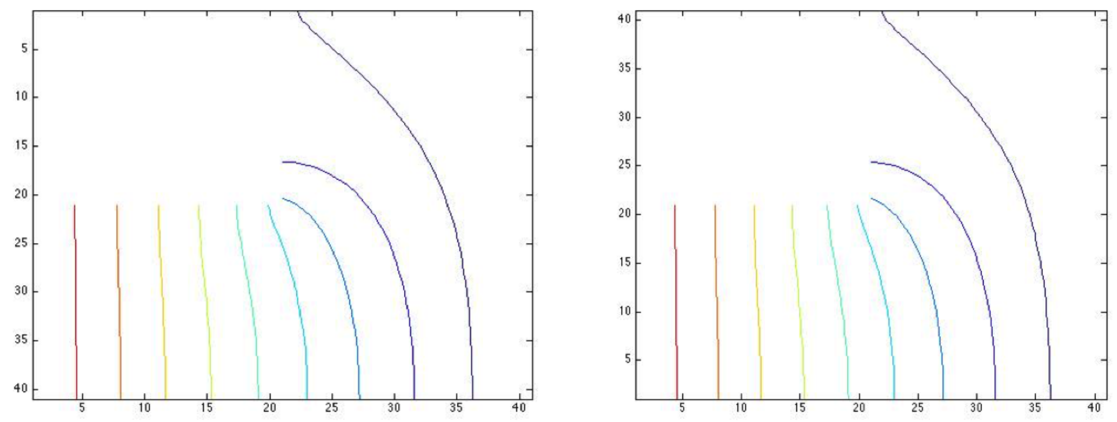
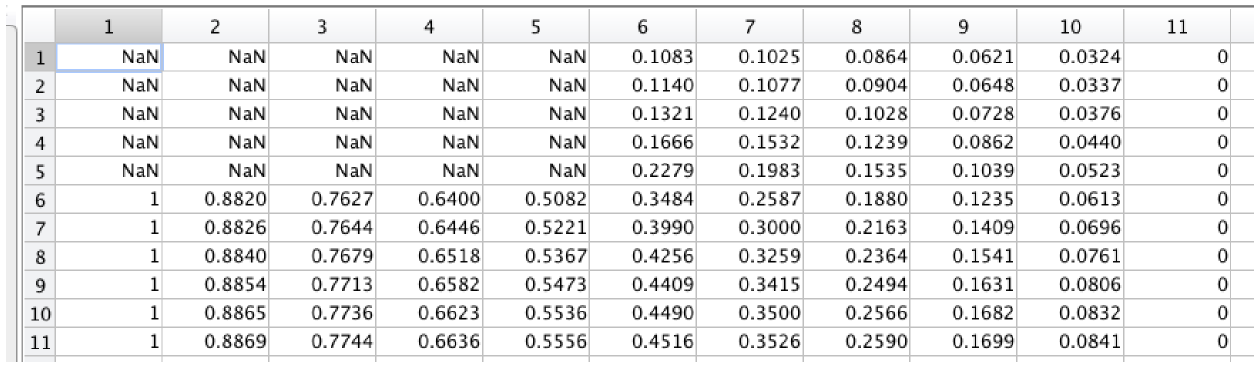
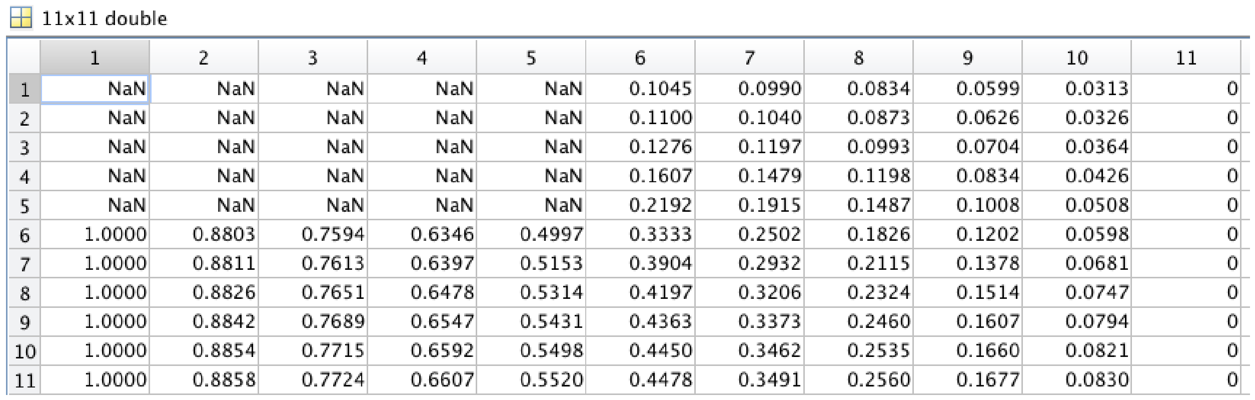


Figure 8. The iso-value lines for FD and FE with h = 0.05

As these figures show, the two results of the two methods are almost the same.



	1	2	3	4	5	6	7	8	9	10	11
1	NaN	NaN	NaN	NaN	NaN	0.1083	0.1025	0.0864	0.0621	0.0324	0
2	NaN	NaN	NaN	NaN	NaN	0.1140	0.1077	0.0904	0.0648	0.0337	0
3	NaN	NaN	NaN	NaN	NaN	0.1321	0.1240	0.1028	0.0728	0.0376	0
4	NaN	NaN	NaN	NaN	NaN	0.1666	0.1532	0.1239	0.0862	0.0440	0
5	NaN	NaN	NaN	NaN	NaN	0.2279	0.1983	0.1535	0.1039	0.0523	0
6	1	0.8820	0.7627	0.6400	0.5082	0.3484	0.2587	0.1880	0.1235	0.0613	0
7	1	0.8826	0.7644	0.6446	0.5221	0.3990	0.3000	0.2163	0.1409	0.0696	0
8	1	0.8840	0.7679	0.6518	0.5367	0.4256	0.3259	0.2364	0.1541	0.0761	0
9	1	0.8854	0.7713	0.6582	0.5473	0.4409	0.3415	0.2494	0.1631	0.0806	0
10	1	0.8865	0.7736	0.6623	0.5536	0.4490	0.3500	0.2566	0.1682	0.0832	0
11	1	0.8869	0.7744	0.6636	0.5556	0.4516	0.3526	0.2590	0.1699	0.0841	0



11x11 double

	1	2	3	4	5	6	7	8	9	10	11
1	NaN	NaN	NaN	NaN	NaN	0.1045	0.0990	0.0834	0.0599	0.0313	0
2	NaN	NaN	NaN	NaN	NaN	0.1100	0.1040	0.0873	0.0626	0.0326	0
3	NaN	NaN	NaN	NaN	NaN	0.1276	0.1197	0.0993	0.0704	0.0364	0
4	NaN	NaN	NaN	NaN	NaN	0.1607	0.1479	0.1198	0.0834	0.0426	0
5	NaN	NaN	NaN	NaN	NaN	0.2192	0.1915	0.1487	0.1008	0.0508	0
6	1.0000	0.8803	0.7594	0.6346	0.4997	0.3333	0.2502	0.1826	0.1202	0.0598	0
7	1.0000	0.8811	0.7613	0.6397	0.5153	0.3904	0.2932	0.2115	0.1378	0.0681	0
8	1.0000	0.8826	0.7651	0.6478	0.5314	0.4197	0.3206	0.2324	0.1514	0.0747	0
9	1.0000	0.8842	0.7689	0.6547	0.5431	0.4363	0.3373	0.2460	0.1607	0.0794	0
10	1.0000	0.8854	0.7715	0.6592	0.5498	0.4450	0.3462	0.2535	0.1660	0.0821	0
11	1.0000	0.8858	0.7724	0.6607	0.5520	0.4478	0.3491	0.2560	0.1677	0.0830	0

Figure 9. The comparsion of the solutions for FE and FD with same number of elements, the upper one is for FD and the below one is for FE.

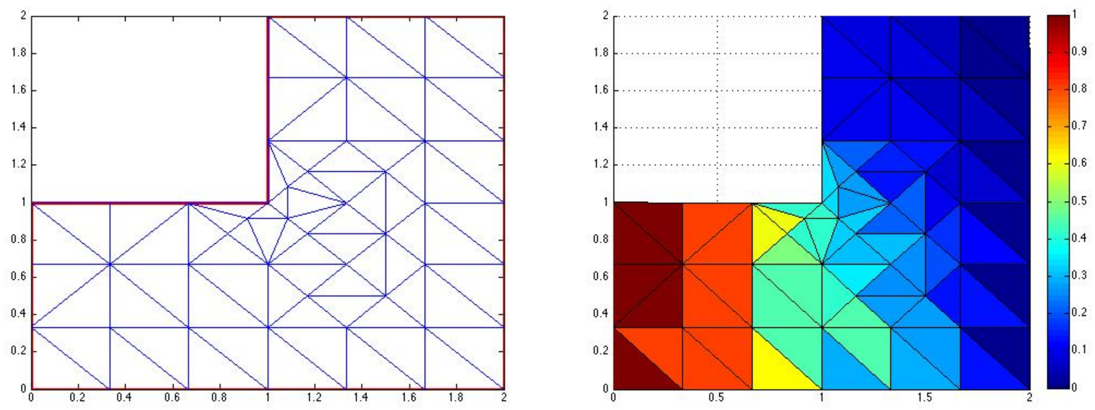
As Figrue 9 shows, the two solutions are nealy the same. Hence, FE and FD are both effective approaches.

4. At first, read the data file using command **textread('point.text')** and there are triangles formed outside the L-shape in the upper left quadrant. Then I used the following commands to remove thses triangles.

```
DT = delaunayTriangulation(P1, C);
IO = isInterior(DT);
hold on
triplot(DT(IO, :),DT.Points(:,1), DT.Points(:,2),'LineWidth', 1);
hold off
```

The command **isInterior** is to find the triangles inside the L-shape. And the command **triplot** is to visualize these triangles.

Using the functions above again, the temperatures of these points can be approximated. The triangles and the visualization of results using the FE method are shown in Figure 10. I found the solutions are not smooth especially in the center corner of the L-shape region where the meshs are unstructured. The reason is that the meshs are sparse and these in the center corner are unstructured which leads to most errors. Hence, I think the most errors occur in the center region and I added points everywhere especially in region around the center corner. I repeat the process again and agin, the new meshs and visualizations are shown from Figure 11 to Figrue 14.



Figrue 10.

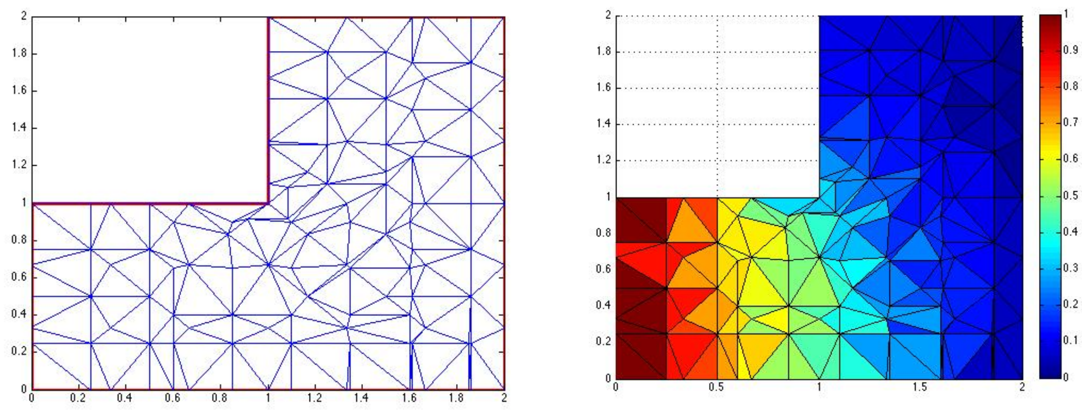


Figure 11.

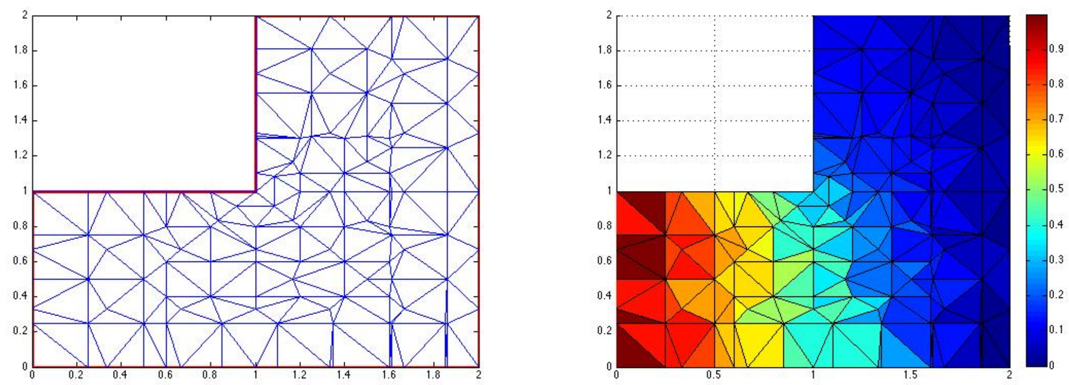


Figure 12.

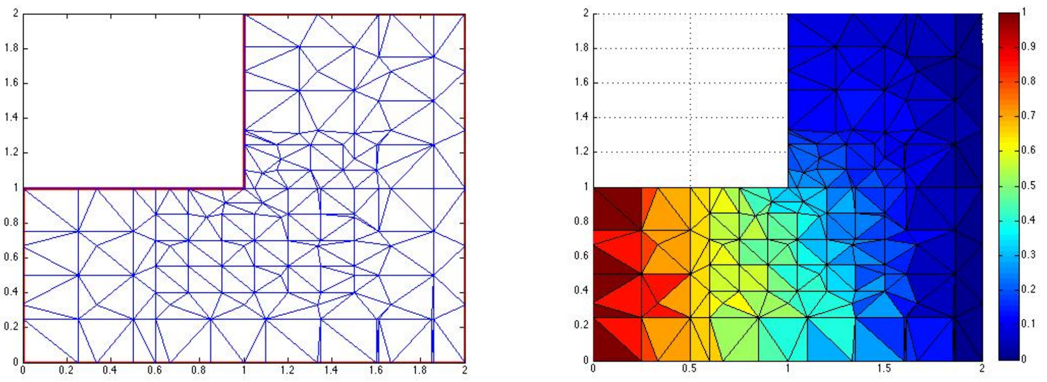


Figure 13.

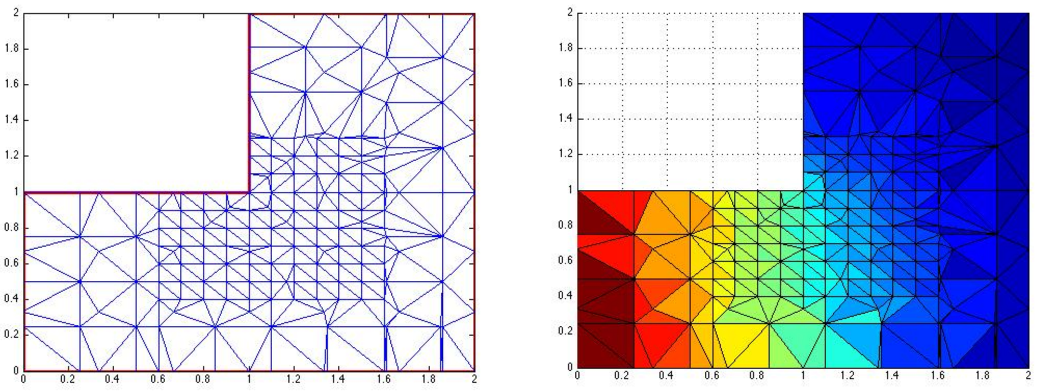


Figure 14.

As the mesh in Figure 11 shows, I added points everywhere uniformly at first and then I stop adding points outside the center region and repeat adding points the center region as the meshs form Figure 12 to Figure 14 show. As a result, the solution becomes more and more accurate and the major error is reduced significantly. I judge the accuracy of solution by observing the visualizations. I implemented the visualization by using command - **trisurf(DT_I,DT.Points(:,1), DT.Points(:,2), res)**. Here, res is solution of Finite Element Method. The colors in the visualization can represent the temperatures or values of points or triangles as the color bar in the figures show. The true result of this heat problem should seem like the visualization of Figure 5. The colors should have a steap tread that changes from dark red to dark blue graduately and smoothly. As Figure 10 to Figure 14 show, the visualization becomes more and more smooth. And the trend becomes more and more obvious. Especially, the trend can't be capatured in the center region as Figure 10 shows at first. After adding points, the colors in that region becom smooth and the trend becomes obvious and steep. The visualization of Figure 14 is close to that of Figure 5, hence, the most error has been reduced and the solution has been more accurate.