

Performance valuation of 2D plane stress elements to capture the displacement and stress in a cantilever beam under static concentrated load

I certify that this submission is my original work and meets the faculty's expectation of originality

Nasibeh Mirvakili

Id 26539769

17/04/2019

***Special thanks to my dear friend Mr. Jamiel
Rahel Id 27321937
without whom I could not submit the
MATLAB part.***

The objective of this project is to compare the results obtained by MATLAB code and also Ansys for displacement and stress at two points of a 2-D cantilever beam with a concentrated load at the end. The effect of shape, size and number of elements on the results are going to be evaluated. Beam is solved with different methods and the results are compared. Displacement and stress are going to be obtained. First, I solve the beam by the equations of the deflection, then I solved the beam by using CST method, applying only two elements. I used Ansys and used different element sizes to get the results, and finally I used MATLAB code to get the results. The objective of this project is to compare the effect of type, and size (number) of elements are used in calculating displacement and stress by MATLAB and Ansys Software.

Introduction

Finite element method is an accurate method for calculating displacement, reaction forces, and stresses in different problems including beams. There are many applications in which we need to find accurate displacement, stress or reaction forces of each segment. By both Ansys software and MATLAB, we are able to find the desired node's displacement. Ansys is able to solve a beam by different methods; including triangular and quadrilateral elements and 2 nodes beam. In triangular we divide the beam into desired number of triangles; we can use three nodes constant strain triangular (CST), or 6 nodes linear strain triangular (LST) elements. In quadrilateral method, we can use 4 nodes quadrilateral (Q4) or 8 nodes quadrilateral (Q8). Q4 means we have an element with four nodes at its 4 corners. Q8 has 4 nodes at its four corners beside 4 nodes in the middle of its four edges. Likewise, CST has 3 nodes on its 3 corners and finally, LST has 3 nodes on its 3 corners beside 3 nodes in the middle of its three edges. In this project elements of 2x8, 4x16, and 8x32 were used, and it means that our object (beam) is divided by 2x8, 4x16, and 8x32 elements each time. In this project effect of type and shape of the element was investigated. Generally, LST, Q8 give better results (more accurate) than CST, and Q4. CST tends to show the least accuracy. Although, Q4 shows lower accuracy than LST, in some applications like stress concentration calculation it might be preferable to LST because of its shape.

Methodology

First of all, I obtained the stress and deflection by equations of deflection and stress for a beam. For the Ansys part of this project, first I modeled the beam as a 2 nodes element 188. In this method we need to define the beam with its coordinates of its nodes cross section. It shows the beam as a line, we put the boundary conditions (no deflection at the node 1 and a load of vertical -2000N at the node 2), and find the stress and deflection at fixed part and free part of the beam. Then, I increased the number of elements until I get an optimum number of elements. In the next step, I used solid plane182,183. Plane 182 is used for CST and Q4; plane 183 is used for LST and Q8. In each method we need to define thickness of the beam (0.05 m). We divide the beam to our desired elements (triangular, quadrilateral), simply we divide each line into the number of elements and obtain the type and number of elements we need. For example, if we need to use CST 2x8, we use plane 182, and in the meshing part we choose element triangle, and

split the top and lower part to 8 and the two remaining parts to 2. So, we will have 2x8 CST elements. Next, we refine the mesh by adding the number of elements. This time we use 4x16 elements by using the same procedure. And finally, we get the results by increasing the number of elements to 8x32. For Q8 and LST, we choose solid element of 183, after defining properties of the material, we model the beam as a plate with a thickness of 0.05 m. In the meshing part, we define the type of our element (triangle for LST and rectangle for Q8), same as before, we divide the upper and lower lines to 8, and the two remaining lines to 2, so we get 2x8 elements. By applying boundary condition, we get our results. In MATLAB part, we wrote the code which find B matrix, and stiffness matrix. In the code we, provide a triangular element of CST, with three nodes on its three corners.

When we have 4x16 elements, we get 128 elements and 85 nodes. Since each node has 2DOF, we need a stiffness matrix of size 170x170.



Figure 1. LST, CST element (2x8 elements)

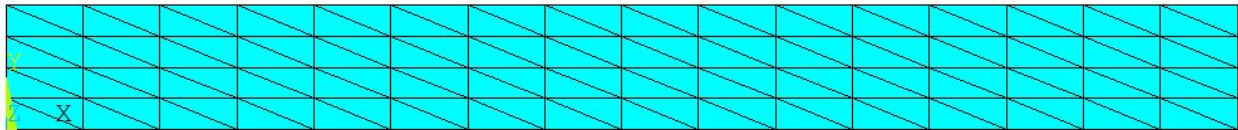


Figure 2. LST, CST element (4x16 elements)

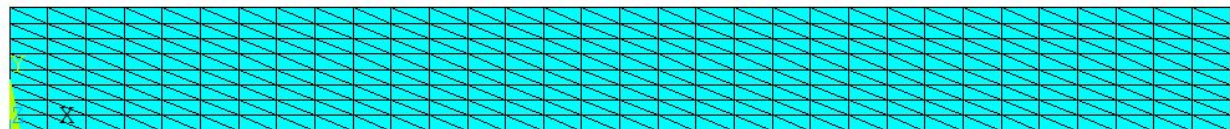


Figure 3. LST, CST element (8x32 elements)

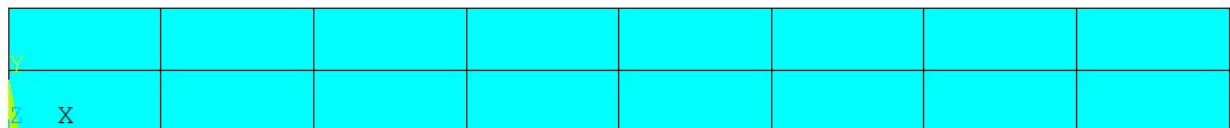


Figure 4. Q4, Q8 element (2x8 elements)

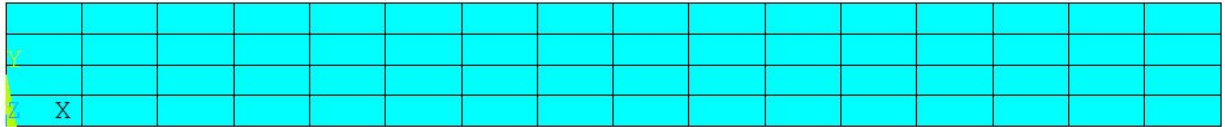


Figure 5. Q4, Q8 element (4x16 elements)












Element Order	2D Solid	3D Solid	3D Shell	Line Elements
Linear	 PLANE42 PLANE182	 SOLID45 SOLID185	 SHELL63 SHELL181	 BEAM3/44  BEAM188
Quadratic	 PLANE82/183  PLANE2	 SOLID95/186  SOLID92/187	 SHELL93 <div style="text-align: right; font-size: small;">Belcan Engineering Group, Inc</div>	 BEAM189

Figure 6. triangular and quadrilateral elements.

First of all, we solve the beam by equation of deflection of the beam:

$$\delta = pl^3/3EI$$

$E=210 \times 10^9$, $I = (1/12) \times (0.5^3) \times (0.05) = 0.521 \times 10^{-3} \text{m}^4$, $l=5 \text{ m}$, $p=2000 \text{ N}$

$$\delta = \frac{pl^3}{3EI} = 0.7612 \text{ mm}$$

$$\sigma_B = MC/I = 2000 \times 0.25 / 0.521 \times 10^{-3} = -959.7 \text{ Mpa}$$

Then we solve the problem by method we learnt in FE, so we split the beam to two triangular element

$$[B1] =$$

-0.5000	0	0.5000	0	0	0
0	0	0	-5.0000	0	5.0000
0	-0.5000	-5.0000	0.5000	5.0000	-0.5000

$$[K1] = [B]^T[D][B]$$

$$D = 210 \times 10^9 / 0.91 \times$$

1.0000	0.3000	0
0.3000	1.0000	0
0	0	0.3000

$$[B]^T =$$

-0.5000	0	0
0	0	-0.5000
0.5000	0	-5.0000
0	-5.0000	0.5000
0	0	5.0000
0	5.0000	-0.5000

$$[K2] = 7.21 \times 10^9$$

0.2500	0	-0.2500	0.7500	0	-0.7500
0	0.0750	0.7500	-0.0750	-0.7500	0.0750
-0.2500	0.7500	7.7500	-1.5000	-7.5000	1.5000
0.7500	-0.0750	-1.5000	25.0750	0.7500	-25.0750
0	-0.7500	-7.5000	0.7500	7.5000	-0.7500

-0.7500 0.0750 1.5000 -25.0750 -0.7500 25.0750

[K1]= 7.21 x10⁹

7.5000	0	0	-0.7500	-7.5000	0.7500
0	25.0000	-0.7500	0	0.7500	-25.0000
0	-0.7500	0.2500	0	-0.2500	0.7500
-0.7500	0	0	0.0750	0.7500	-0.0750
-7.5000	0.7500	-0.2500	0.7500	7.7500	-1.5000
0.7500	-25.0000	0.7500	-0.0750	-1.5000	25.0750

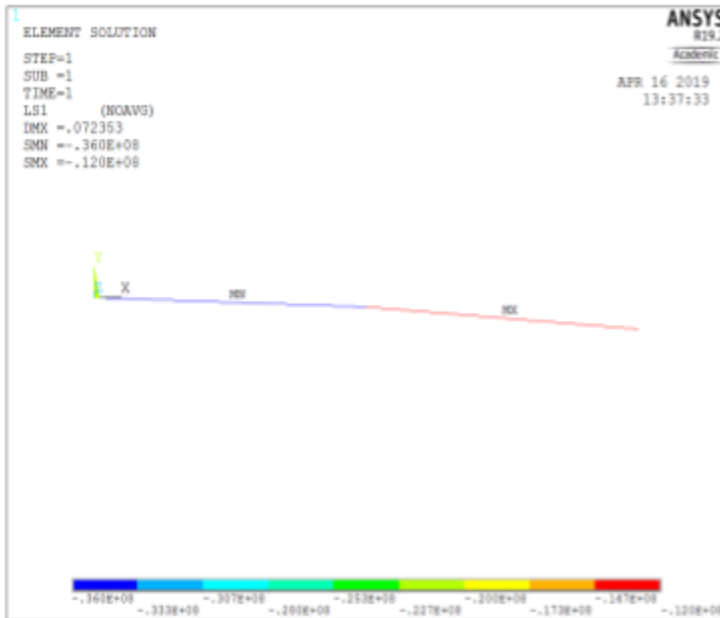
[K_{sys}]=7.750 0 -7.500 0.7500 0 -1.500 -0.2500 0.7500

0	25.0750	0.7500	-25.00	-1.500	0.250	0.7500	0.0750
0	-0.7500	0.2500	0	-0.2500	0.750	0	0
7.750	0.250	1.500	7.500	-0.2500	1.500	0	0
0.250	7.750	7.750	-1.50	50.750	25.750	-1.50	-25.0750
0	-1.500	7.50	-0.250	25.750	50.750	7.50	7.505
-0.2500	0.7500	0	0	-1.5	-25.0750	7.74	-1.5
0.750	-0.250	0	-1.5	0	-25.0750	-1,5	25.0750

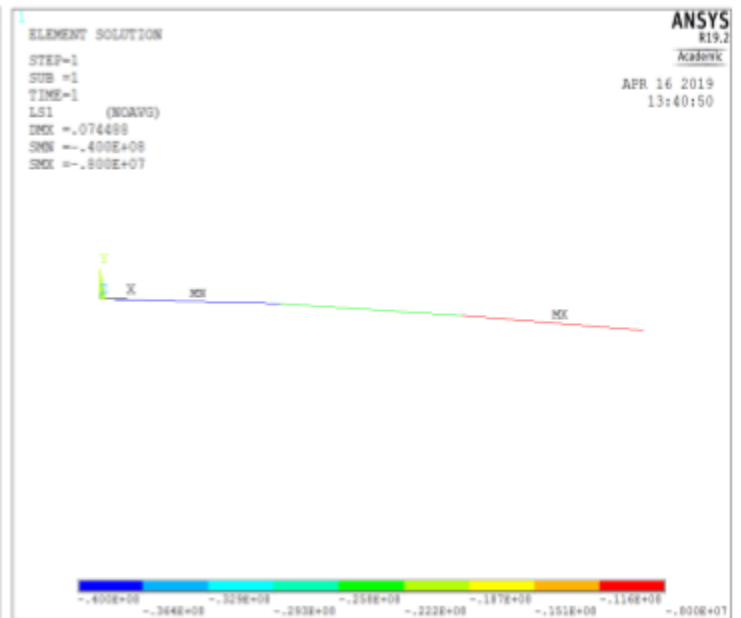
Form this method I get v= 0.643mm

Stress= 18.6 Mpa

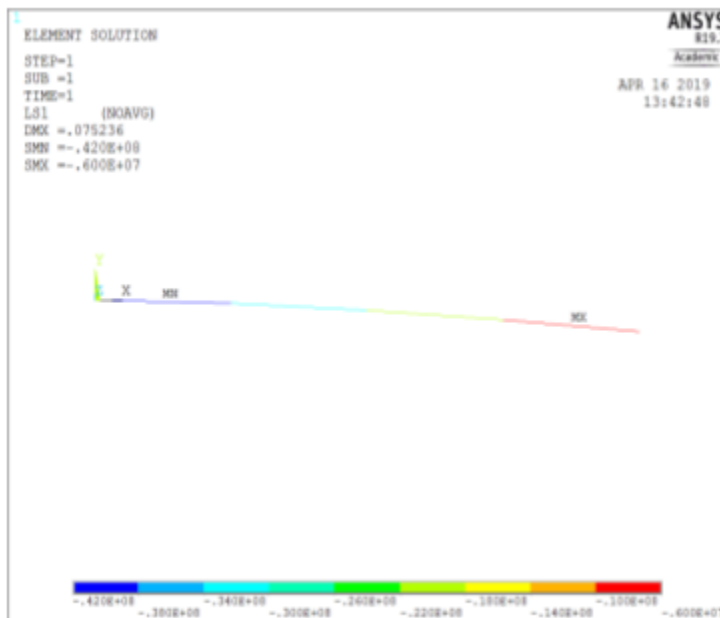
Then, we can see the results from beam element of 18



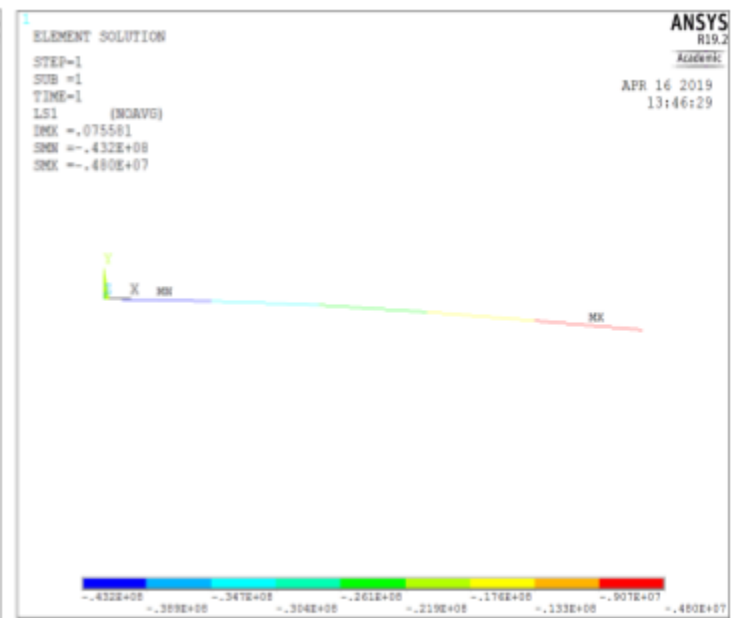
2Nodes



3Nodes



4Nodes



5Nodes

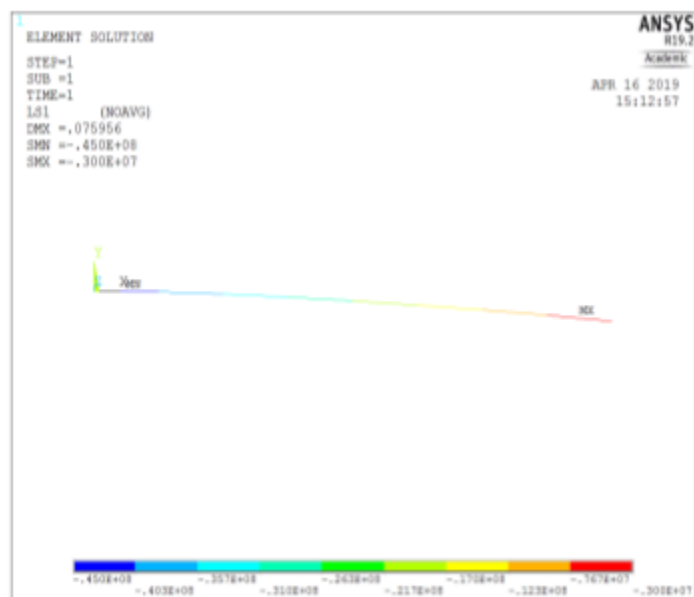
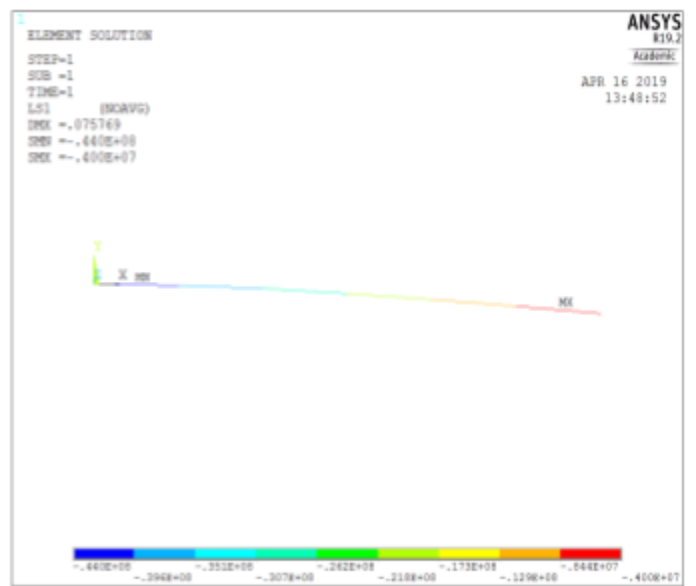
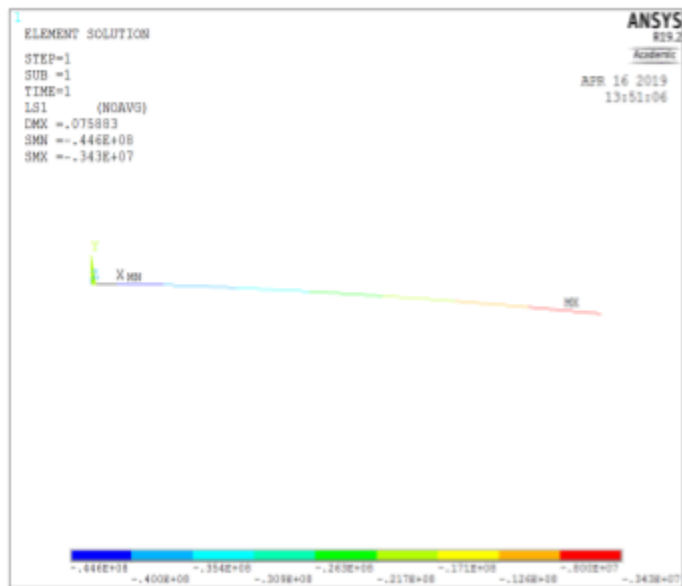


Figure 6- stress and deflection of beam (element 2 nodes,188)

To find the optimum number of a beam we find the percentage error between our results:

After applying 4 nodes, our results begin to converge, so we take the percentage error after 4 nodes

$$\text{Percentage error (between 4 and 5 nodes - displacement)} = \frac{0.075581 - 0.075236}{0.075581} = 0.0046$$

Which is only 0.45 %

So, the optimum element for the beam to get a reasonable result is 4 elements. Obviously, if we apply more elements we get more accurate results and less error. Let's try one more result:

This time we get between 7 and 8 nodes:

$$\text{Percentage error (between 7 and 8 nodes - displacement)} = \frac{0.075956 - 0.075788}{0.075956} = 0.0022$$

Which is 0.22 % error. It is very small but since the previous error (4 nodes) is fairly low, so the optimum element is 4

CST, LST, Q4 and Q8 elements

Stress in X direction

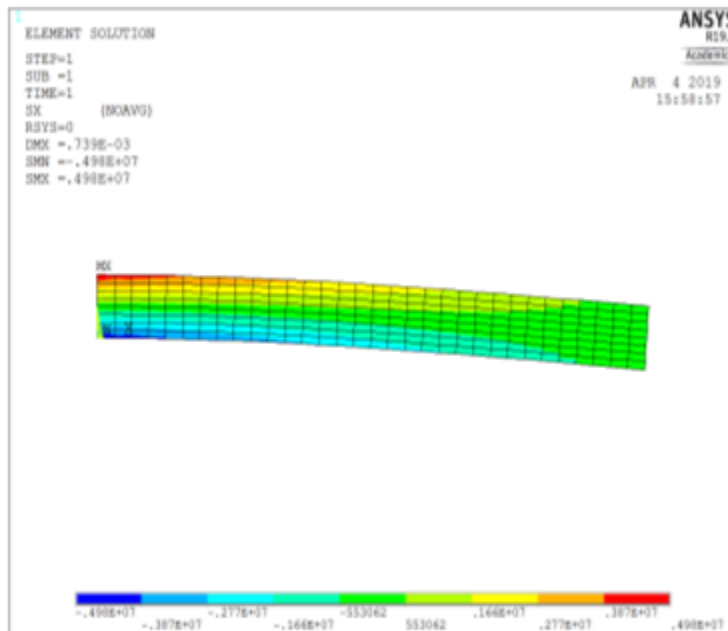
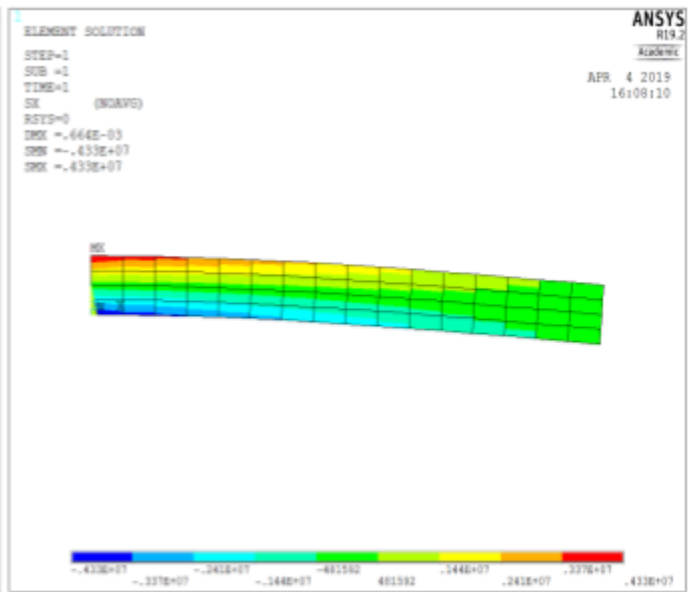
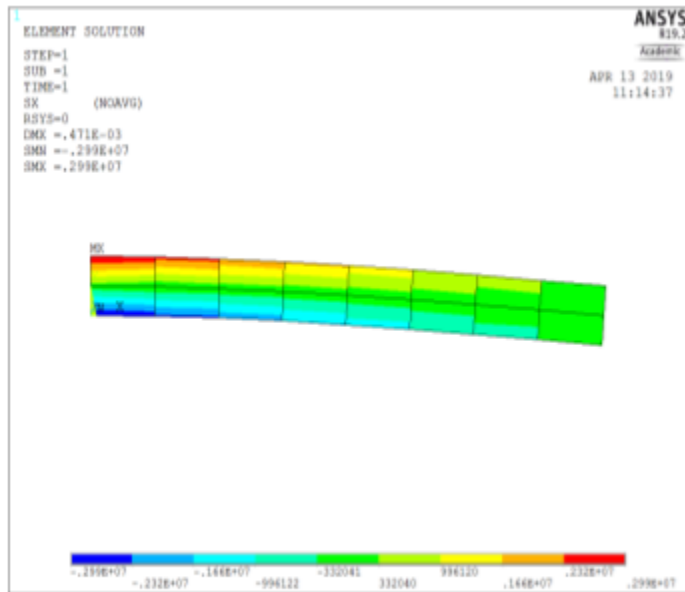
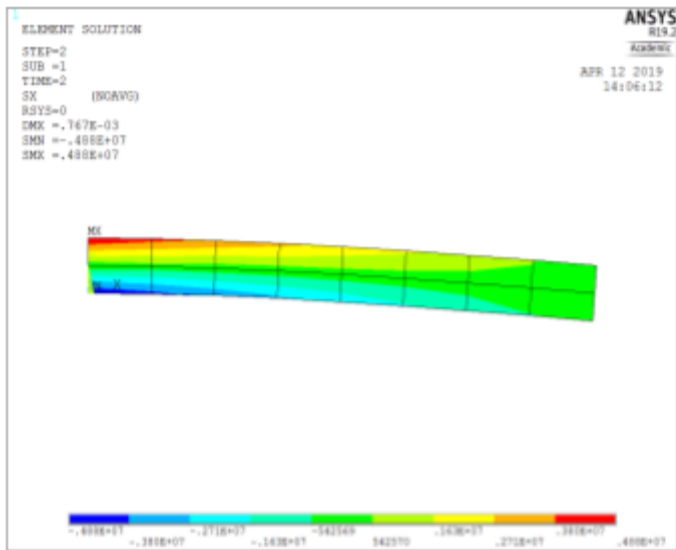
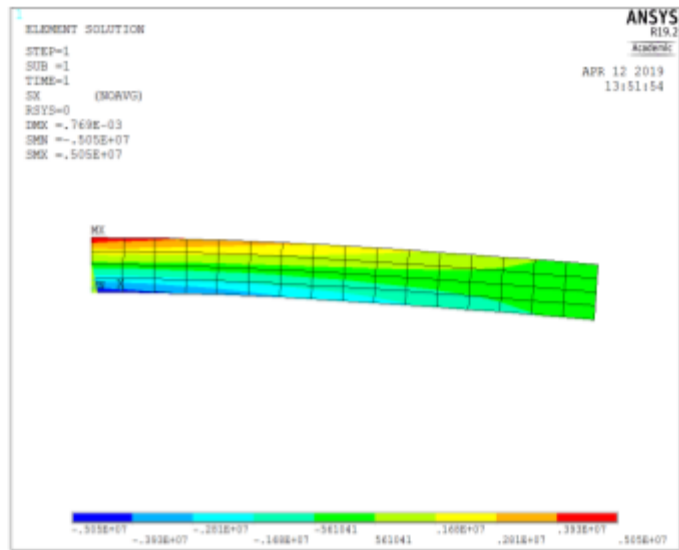


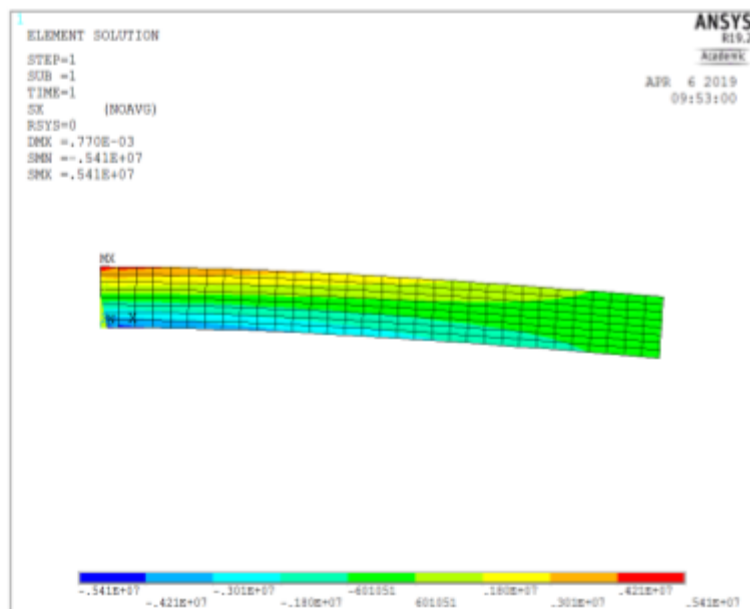
Figure 7- stress in x direction, element Q4 (mesh sizes 2x8,4x16, and 8x3)



Q8-2x8

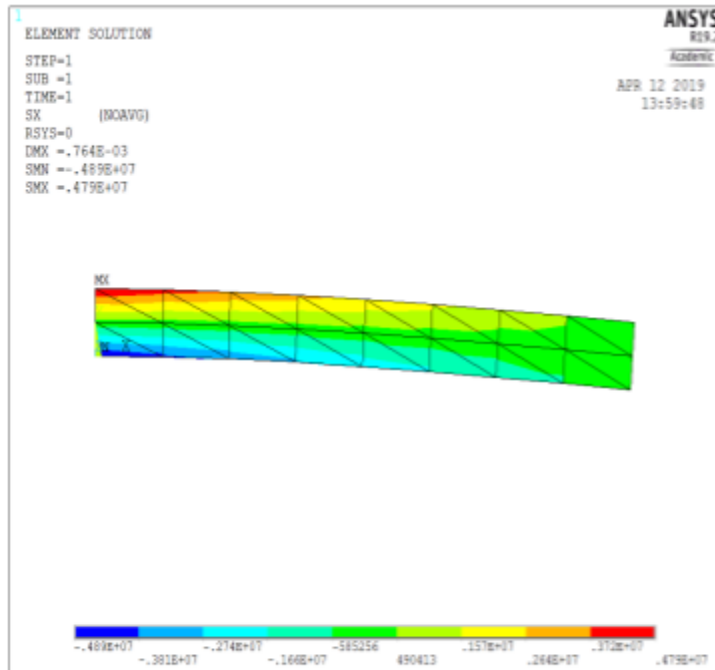


Q8-4x16

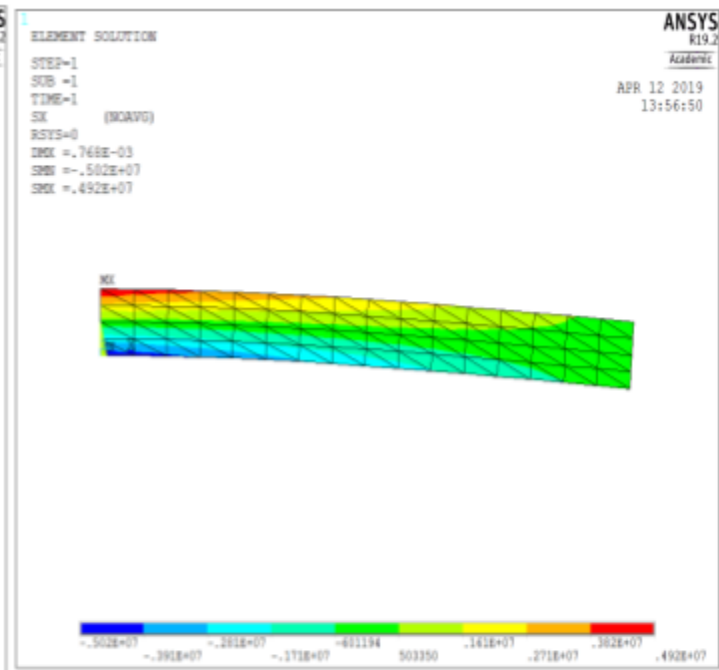


Q8-8x32

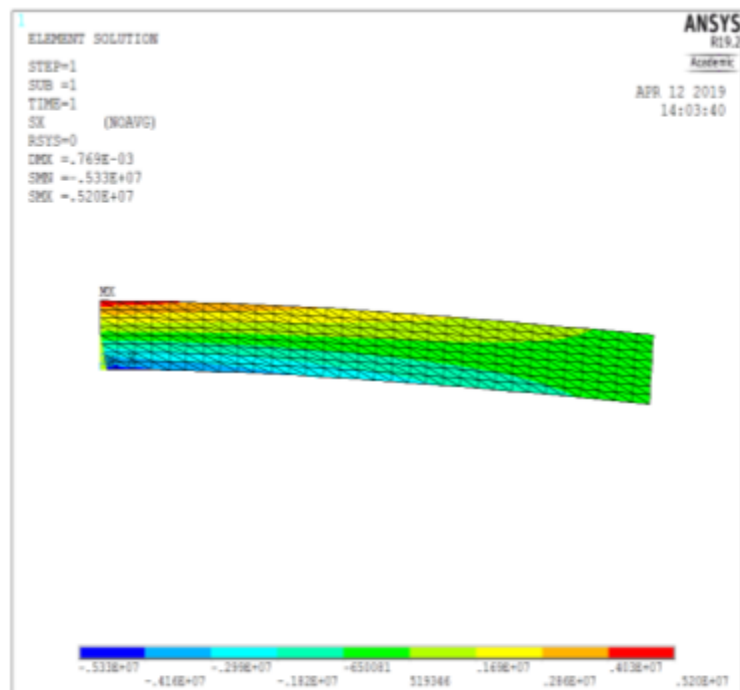
Figure 8- stress in x direction, element Q8 (mesh sizes 2x8,4x16, and 8x32)



LST-2x8

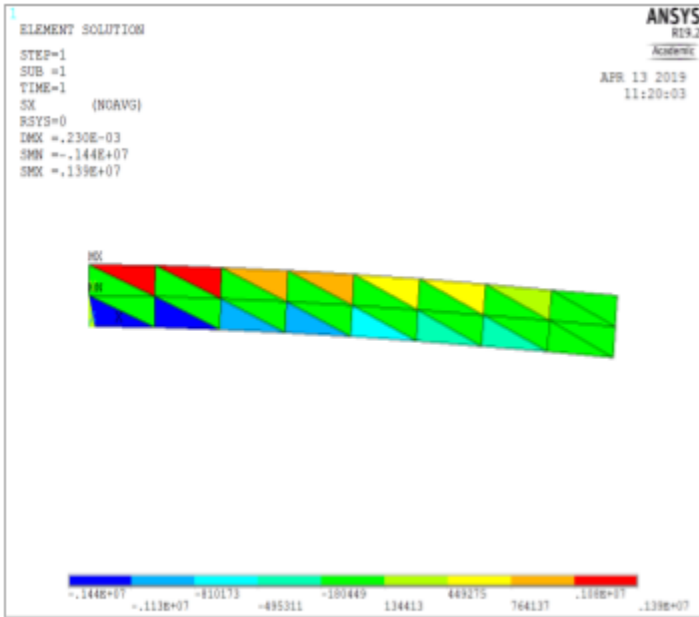


LST-4x16

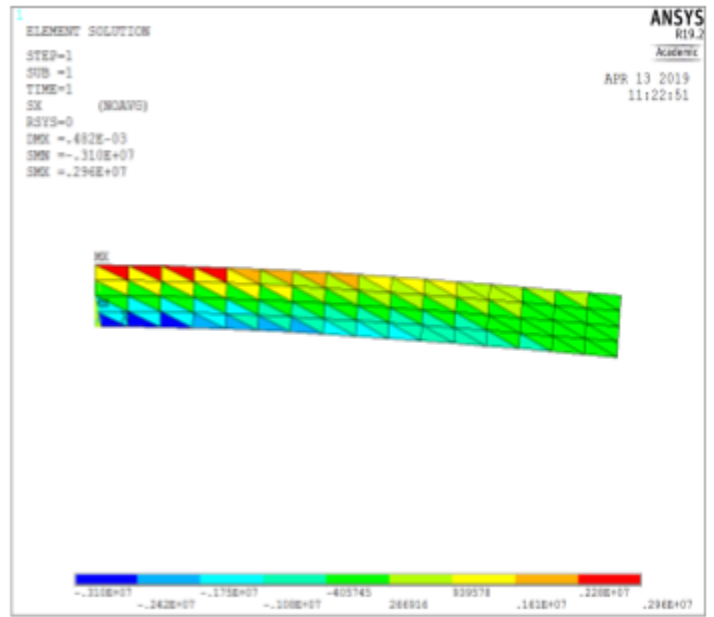


LST-8x32

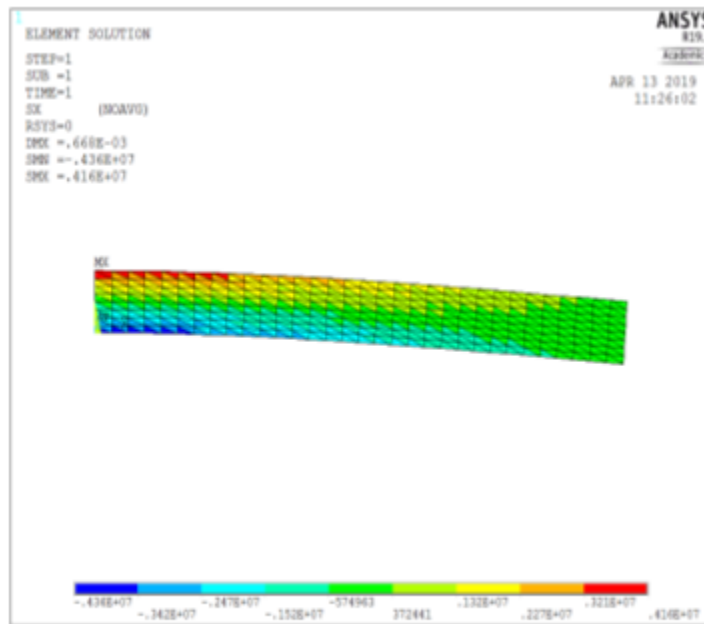
Figure 9- stress in x direction, element LST (mesh sizes 2x8,4x16, and 8x32)



CST 2x8



CST 4x16

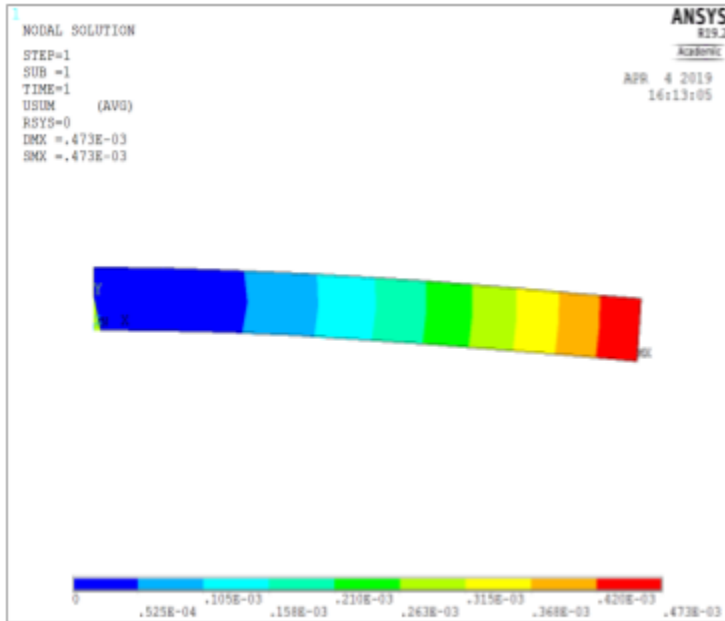


CST 8x32

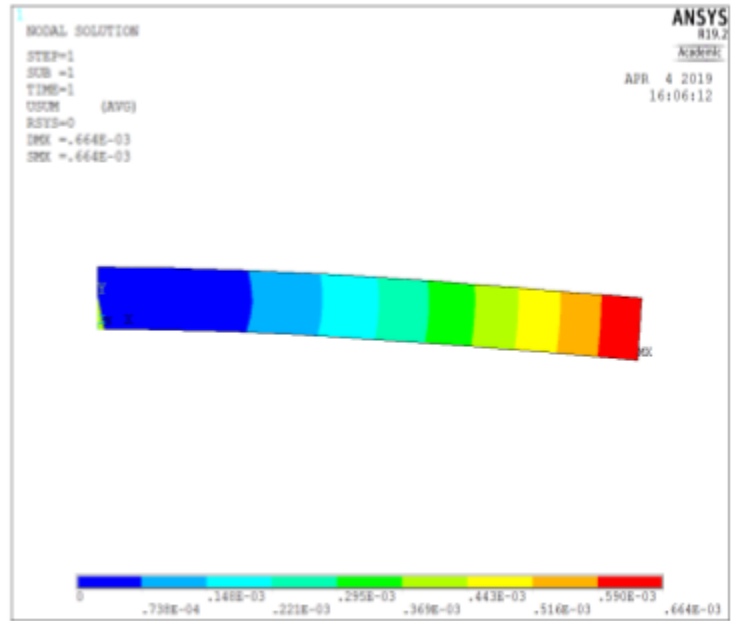
Figure 10- stress in x direction, element CST (mesh sizes 2x8,4x16, and 8x32)

Displacements

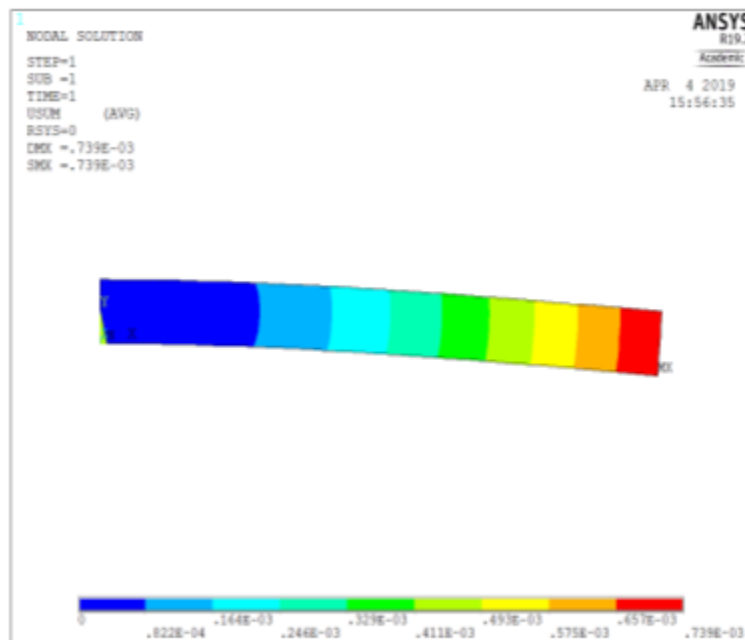
Element Q4,Q8,CST, LST



Q4- 2x8

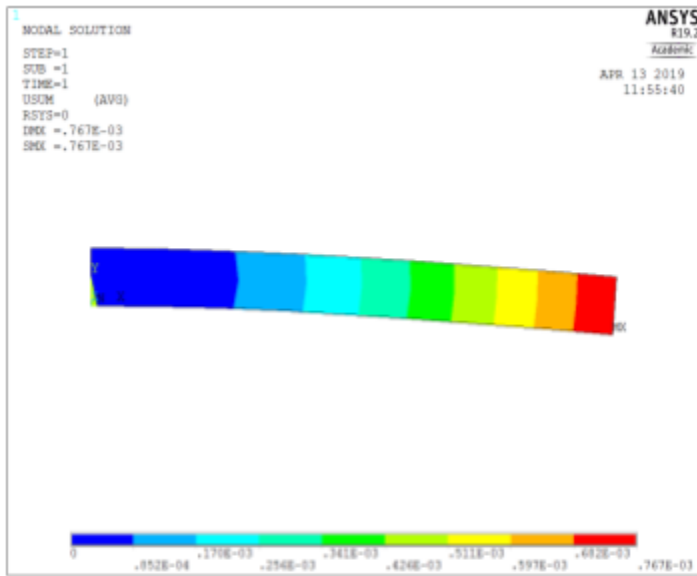


Q4- 4x16

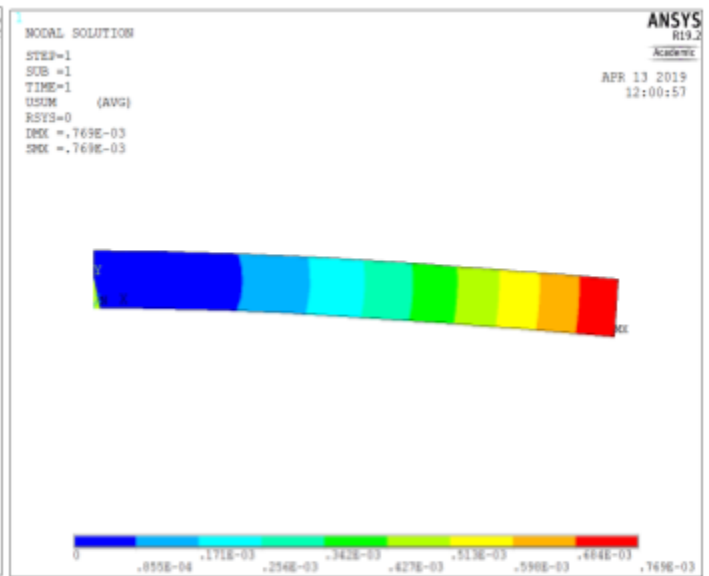


Q4- 8x32

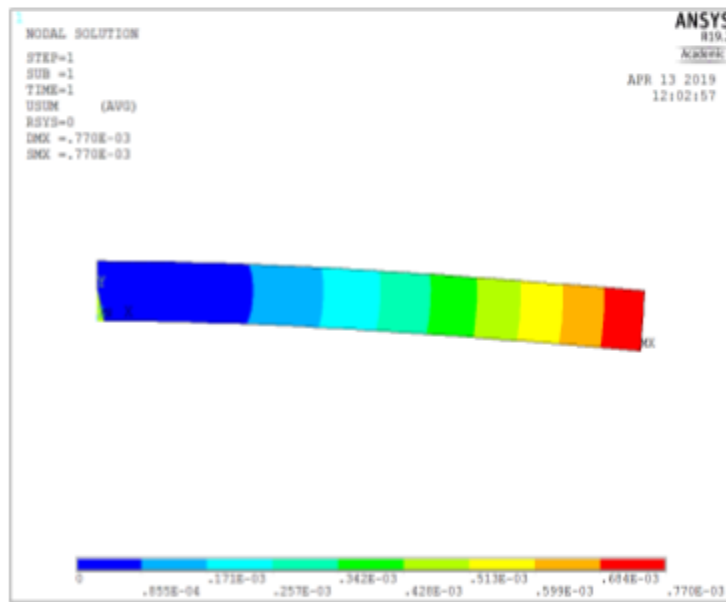
Figure 11- displacements, element Q4 (mesh sizes 2x8,4x16, and 8x32)



Q8- 2x8

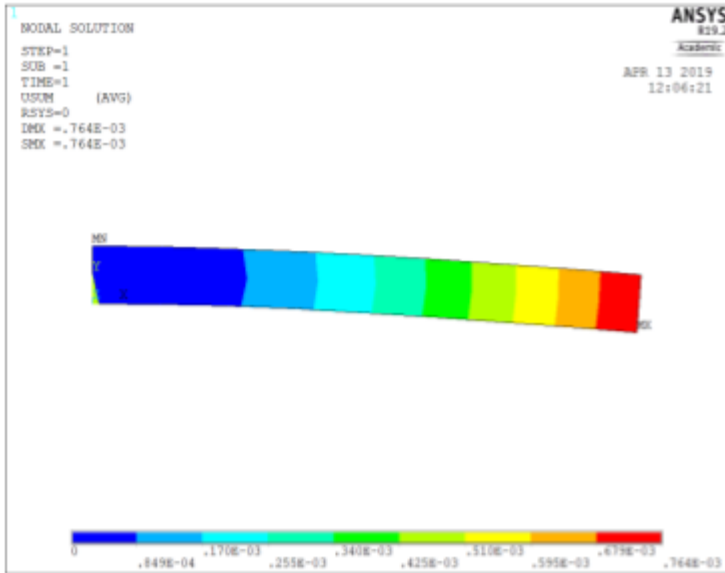


Q8- 4x16

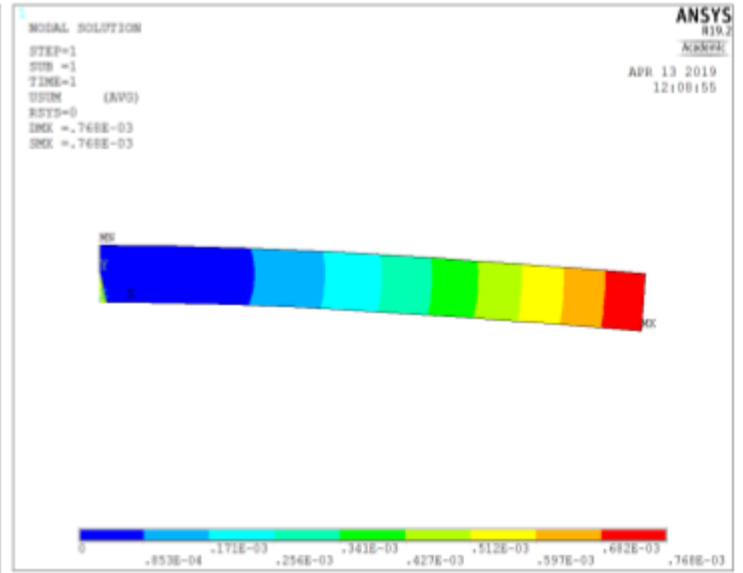


Q8- 8x32

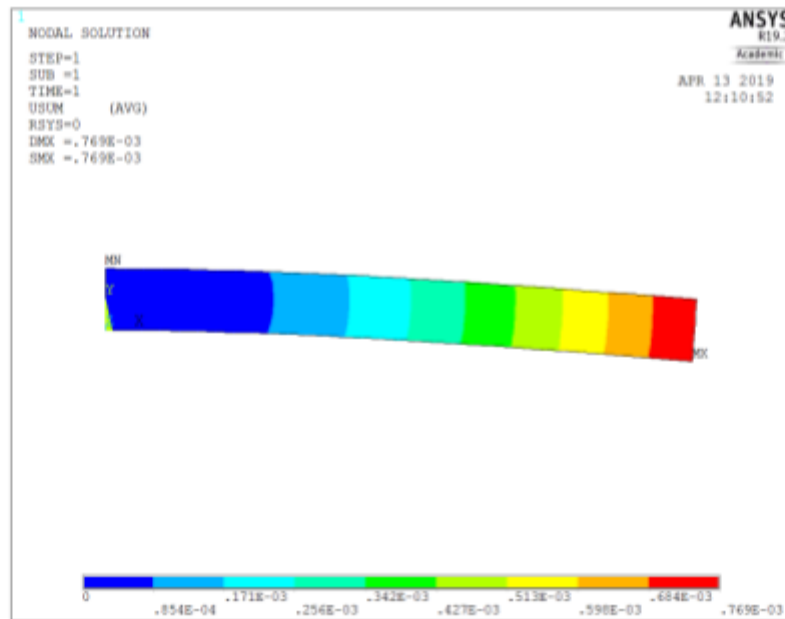
Figure 12- displacements, element Q8 (mesh sizes 2x8,4x16, and 8x32)



LST- 2x8

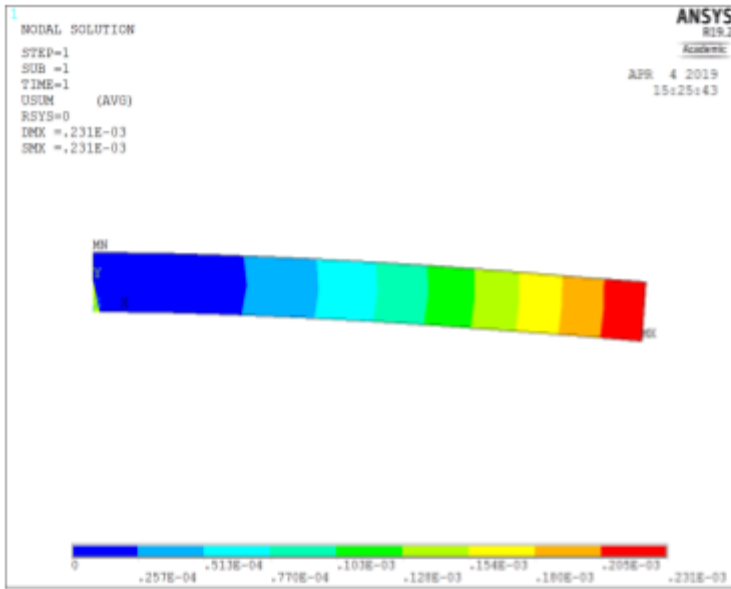


LST- 4x16

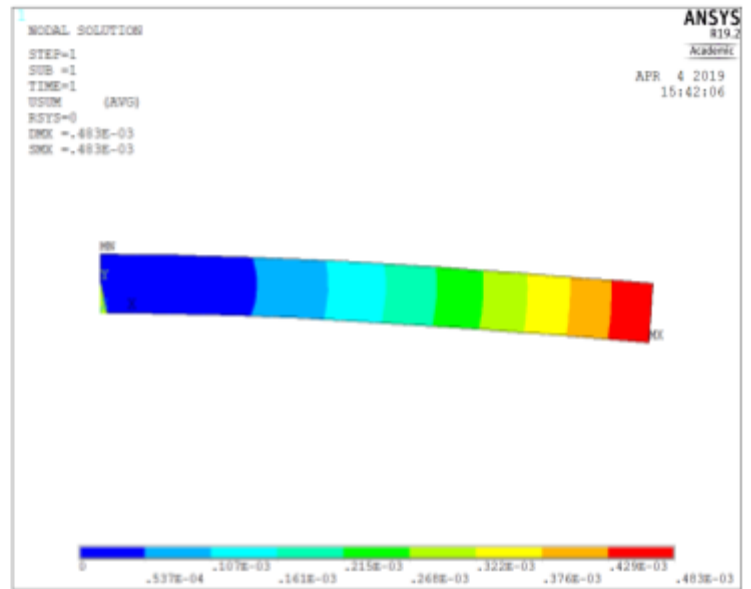


LST- 8x32

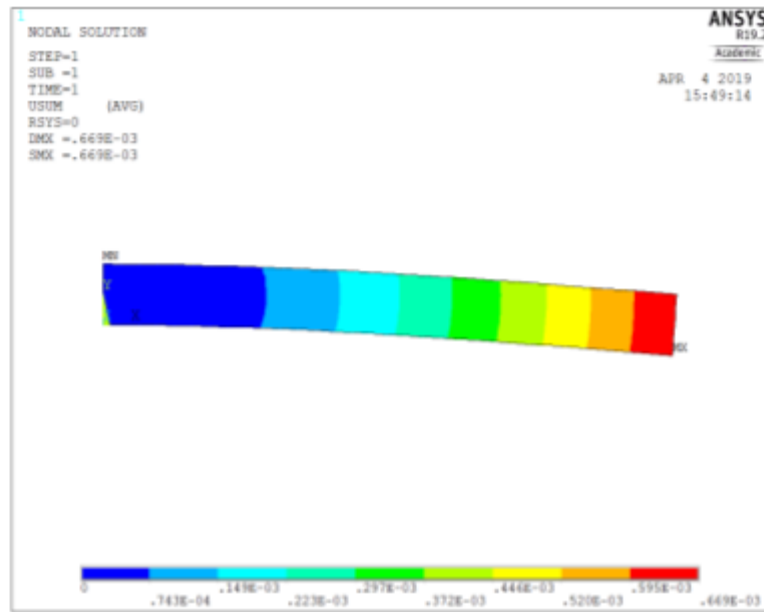
Figure 13- displacements, element LST (mesh sizes 2x8,4x16, and 8x32)



CST- 2x8



CST- 4x16



CST- 8x32

Figure 14- displacements, element CST (mesh sizes 2x8,4x16, and 8x32)

ANSYS part			MATLB	
	<i>Displacement at A (mm)</i>	<i>X direction Stress x10⁷ pa</i>	<i>Displacement mm</i>	<i>x -stress x10⁷</i>
CST-2x8	0.231	0.145		
CST-4x16	0.483	0.31		
CST-8x32	0.669	0.37		
LST-2x8	0.764	0.488		
LST-4x16	0.768	0.502		
LSt-8x32	0.769	0.533		
Q4-2x8	0.473	0.3		
Q4-4x16	0.664	0.433		
Q4-8x32	0.739	0.498		
Q8-2x8	0.767	0.488		
Q8-4x16	0.769	0.505		
Q8-8x32	0.77	0.541		
Beam-2nodes	0.072353	3.6	0.640	
Beam 3 nodes	0.074488	0.4	0.653	
Beam 4nodes	0.075236	4.2	0.662	
Beam 5 nodes	0.075581	4.32	0.668	
Beam 6 nodes	0.075769	4.4	0.681	
Beam 7 node	0.075788	4.46	0.698	
Bam 8 nodes	0.075956	4.5	0.702	
Equations of beams	0.71612	0.956		

Table 1- comparison of the stress and deflections from MATLB, Ansys, and equations of deflections of a beam

Discussion

Generally, LST and Q8 give more accurate results for displacement because they have nodes in the middle of their edges. But for stress we get a better result by element Q4. CST shows the least accurate results.

As we can see in the results, LST and Q* provide almost the same results.

Percentage error (between LST and Q8 -8x32 -displacement) = $0.770 - 0.769 - 0.770 = 0.0013$

0.13%

If we get the error for 4x16- element Q8, LST, we find:

Error = $0.769 - 0.769 / 0.769 = 0.0013$ 0.13% error

So LST and Q8 give the same result for displacement. However, for stress their results from LST 8x32 and Q8 8x32 shows a higher error:

Error = $0.541 - 0.533 / 0.541 = 0.015$ (1.5% error)

Overall, the results from Q4 and LST are close to each other, but CST and Q4 do not show the same result.

CST does not provide accurate result, as we can see the results from CST are far from any other results obtained from the equations of displacement of a beam.

In addition, beam using beam 2 node 188 does not give results close to LST and Q8.

The results have an error of around 90%.

The results from the equation of deflection of beam ($\delta = pl^3/3EI$) are close to the results from the triangular and quadrilateral elements.

Moreover, we can see that 2 -node beam 188 provides the least accurate results. Compared to LST, Q4, Q8, the results from beam element 188 is not reliable.

Now let's, find the percentage error from the results obtained by the equations of deflection with Q8, and Q4 results.

Error = $0.769 - 0.71612 / 0.769 = 0.068$ error = 6.8 % (LST8x32 compared to deflection equations)

Error= $0.769 - 0.739 / 0.769 = 0.039$ error = 3.9 % (Q4- 8x32 compared to deflection equations)

Results from MATLAB are closer to SLT and Q4,8 than beam 2-d nodes, however they have an error around

Error= $0.77 - 0.702 / 0.77 = 0.065$ around 6.5% error

Results from MATLAB, are different form ANSYS, by a magnitude of 100 (Ansys provides displacements about 100 timer higher than MATLAB).

Conclusion

As the number of nodes increases, the accuracy of our results increases as well. However, there is a certain number of elements that provide sufficient and reasonable results without consuming extra time. In complex applications, the mount of time and also the capacity of CPU play a great role, so it is important to obtain an optimum amount of element and also use a proper element.

For example, LST and Q8 gives accurate results because they are providing nodes in the middle of their edges, however in some application Q4 is preferable because of nature of its shape.

In addition, despite Q8 and LST provide almost same results, in some application LST might be preferable because it is faster. Results from MATLAB codes and ANSYS are not close.

Appendix

- It's not very efficient due to the heavy use of nested loops. The worst-case complexity is roughly $O(n^2)$, but this may be unavoidable. - In the init script, there is a significant amount of duplication in filling in the nodes. Essentially, most nodes are computed twice. There is room for improvement in this area.

- The class structure is mutually recursive. The node objects contain element objects and vice versa. The purpose of this was to save heavy computations later on when determining the total K matrix. In the other matrix computations (B and K), it is sufficient to use a hierarchy of elements->nodes, where all elements can be iterated over and the nodes contained in those elements used for computation. However, in the total K matrix, it is necessary to know which elements are associated with a given node. This requires the opposite hierarchy of nodes->elements. Instead of laboriously searching the matrix for elements that touch each node, it made more sense to embed that information as it's being collected in init. This increases the space complexity, but only by a constant amount, since a node is never associated with more than 6 elements. - The total K matrix is nearly ready to implement, but was not due to time constraints.

MATLAB Code

Main Script

main.m

```
1  init;
2  getD;
3  getB;
4  getK;
5  getTotalK;
6
7  % Get displacement at node 81 (where the force of 2000N is applied)
8  i = 81*2 - 1;
9  K_sub = K(i:i+1,i:i+1);
10 R = [0;2000];
11 U = linsolve(K_sub, R);
12 disp('Displacement at node 81:')
13 disp(U)
```

Concluding Notes

- Result is 0.1358×10^{-5} meters in the y direction.
- U represents (x,y) displacement of node 81.
- Resulting U is 2 orders of magnitude smaller than ANSYS simulations, but the method used by ANSYS is probably very different.
- The displacement is 0 in the x direction since there is no force in the x direction.
- When a diagonal force is applied, there is more displacement in the x direction than y direction. This is what we expect, which somewhat validates the code. Further validation would have to be done by comparing randomly selected parts of the K matrix with hand-calculated values.

Data Structures

The data structures used in the program are called **Node** and **Element**. At a glance, they appear to be mutually recursive, however MATLAB does not have pointers therefore true mutual recursion is not possible. Thus, an **Element** contains the *indices* of its nodes, and a **Node** similarly contains the *indices* of the elements that it touches.

Node.m

```
1 classdef Node
2     properties
3         x
4         y
5         index
6         elements % indices of the elements
7     end
8 end
```

Element.m

```
1 classdef Element
2     properties
3         node_i % local index 1
4         node_j % local index 2
5         node_m % local index 3
6         B
7         K
8         % map global index (1,3) to local index (1,85)
9         % (containers.Map object)
10        index_map
11        index
12    end
13 end
```

Initialize Elements and Nodes

This script creates the `nodes` and `elements` arrays with the data structures outlined above.

It then fills all the values based on the given system.

`init.m`

```
1 clear all
2 clc
3
4 % Initialize entire mesh
5
6 elements(128) = Element;
7 nodes(85) = Node;
8
9 index = 1;
10 for i=0:15
11     for j=0:3
12         odd = 2*j+1 + 8*i;
13         elements(odd).index = odd;
14
15         % Create node i
16         node_i = Node;
17         node_i.x = i*0.3125;
18         node_i.y = j*1.25;
```

```

19     node_i.index = index;
20
21     % Add node to total node list
22
23     if isempty(nodes(index).elements)
24         nodes(index) = node_i;
25     elseif isempty(nodes(index).x)
26         nodes(index).x = node_i.x;
27         nodes(index).y = node_i.y;
28         nodes(index).index = index;
29     end
30
31     % Append element to the node's list of elements
32
33     nodes(index).elements = [nodes(index).elements, odd];
34
35     % Initialize index map with node i
36
37     % Remember that i → 1, j → 2, m → 3
38
39     elements(odd).index_map = containers.Map({index},{1});
40
41     % Save node i to the element
42
43     elements(odd).node_i = index;
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```



```

41     % Add node to total node list
42     if isempty(nodes(index+1).elements)
43         nodes(index+1) = node_m;
44     elseif isempty(nodes(index+1).x)
45         nodes(index+1).x = node_m.x;
46         nodes(index+1).y = node_m.y;
47         nodes(index+1).index = index+1;
48     end
49     % Append element to the node's list of elements
50     nodes(index+1).elements = [nodes(index+1).elements, odd];
51     % Add node to element(odd)'s index map
52     elements(odd).index_map(index+1) = 3;
53     % Save node m to the element
54     elements(odd).node_m = index+1;
55
56     % Create node j
57     node_j = Node;
58     node_j.x = node_i.x + 0.3125;
59     node_j.y = node_i.y;
60     node_j.index = node_i.index + 5;
61     % Add node to total node list
62     if isempty(nodes(index+5).elements)

```

```

63         nodes(index+5) = node_j;
64     elseif isempty(nodes(index+5).x)
65         nodes(index+5).x = node_j.x;
66         nodes(index+5).y = node_j.y;
67         nodes(index+5).index = index+5;
68     end
69     % Append element to the node's list of elements
70     nodes(index+5).elements = [nodes(index+5).elements, odd];
71     % Add node to element(odd)'s index map
72     elements(odd).index_map(index+5) = 2;
73     % Save node j
74     elements(odd).node_j = index+5;
75
76     even = 2*(j+1) + 8*i;
77     elements(even).index = even;
78
79     elements(even).node_i = elements(odd).node_m;
80     elements(even).node_j = elements(odd).node_j;
81
82     % Create node m for the even element
83     node_m_even = Node;
84     node_m_even.y = nodes(elements(even).node_i).y;

```

```

85     node_m_even.x = nodes(elements(even).node_j).x;
86     node_m_even.index = index + 6;
87     if isempty(nodes(index+6).elements)
88         nodes(index+6) = node_j;
89     elseif isempty(nodes(index+6).x)
90         nodes(index+6).x = node_j.x;
91         nodes(index+6).y = node_j.y;
92     end
93     nodes(index+6).index = index+6;
94     % Fill even element's index map
95     elements(even).index_map = containers.Map({index+1,index+5,index
96         +6},{1,2,3});
97     % Append even element to each node
98     nodes(index+1).elements = [nodes(index+1).elements, even];
99     nodes(index+5).elements = [nodes(index+5).elements, even];
100     nodes(index+6).elements = [nodes(index+6).elements, even];
101
102     % Save node m for the even element
103     elements(even).node_m = index+6;
104
105     index = index + 1;
106 end

```

```
106     index = index + 1;
107 end
108
109 clear node_i node_j node_m node_m_even
110 clear i j odd even index
```

Get B Matrix

getB.m

```
1 % Elements have been initialized with associated nodes
2 % This script obtains the B matrix for each element
3
4 A = (1.25*0.3125)/2; % Area per element
5
6 for e=1:128
7     % Calculate beta i,j,m and gamma i,j,m
8     beta_i = nodes(elements(e).node_j).y - nodes(elements(e).node_m).y;
9     beta_j = nodes(elements(e).node_m).y - nodes(elements(e).node_i).y;
10    beta_m = nodes(elements(e).node_i).y - nodes(elements(e).node_j).y;
11
12    gamma_i = nodes(elements(e).node_m).x - nodes(elements(e).node_j).x;
13    gamma_j = nodes(elements(e).node_i).x - nodes(elements(e).node_m).x;
14    gamma_m = nodes(elements(e).node_j).x - nodes(elements(e).node_i).x;
15
16    % Now construct B matrix for the kth element
17    elements(e).B = (1/(2*A))*[beta_i 0 beta_j 0 beta_m 0;
18                               0 gamma_i 0 gamma_j 0 gamma_m;
19                               gamma_i beta_i gamma_j beta_j gamma_m beta_m];
20 end
```

Get D Matrix

getD.m

```
1 % Get elasticity matrix D for the system
2
3 E = 210e9; % Modulus of elasticity
4 v = 0.33; % Poisson ratio
5
6 c = E/(1-v^2);
7
8 D = c*[1 v 0; v 1 0; 0 0 (1-v)/2];
9
10 clear c
```

Get Element K Matrix

getK.m

```
1 % Get stiffness matrix K for each element
2
3 t = 0.05; % Beam thickness
4
5 for i=1:128
6     B = elements(i).B;
7     B_transpose = elements(i).B.';
8     elements(i).K = t * A * B_transpose * D * B;
9 end
10
11 clear i
```

Get Total K Matrix

getTotalK.m

```
1 K = zeros(85*2, 85*2);
2
3 % Calculate the system stiffness K_sys
4
5 for index=1:2:128
6
7     % Get the diagonal for node_i in the element
8     n1 = nodes(elements(index).node_i);
9     K_sum = getDiagonalKSum(n1, elements);
10    i = n1.index*2 - 1;
11    K(i:i+1,i:i+1) = K_sum;
12
13    % If the element touches the top we need to
14    % do an extra computation
15    if mod(elements(index).node_m,5) == 0
16        n1 = nodes(elements(index).node_m);
17        K_sum = getDiagonalKSum(n1, elements);
18        i = n1.index*2 - 1;
19        K(i:i+1,i:i+1) = K_sum;
20    end
```



```

21
22 % Special case of the last column where we need to
23 % count node_j for each element
24 if index > 120
25     n1 = nodes(elements(index).node_j);
26     K_sum = getDiagonalKSum(n1, elements);
27     i = n1.index*2 - 1;
28     K(i:i+1,i:i+1) = K_sum;
29 end
30
31 % Very special case of the last odd element 127
32 % We need to include the very last node 85
33 if index == 127
34     n1 = nodes(elements(index+1).node_m);
35     K_sum = getDiagonalKSum(n1, elements);
36     i = n1.index*2 - 1;
37     K(i:i+1,i:i+1) = K_sum;
38 end
39
40 % Horizontal lines
41 % nodes i-j
42 n1 = nodes(elements(index).node_i);

```

```

43     n2 = nodes(elements(index).node_j);
44     [e1,e2] = getTouchingElements(n1,n2);
45     K_sum = getNonDiagonalKSum(n1,n2,e1,e2,elements);
46     i = n1.index*2 - 1;
47     j = n2.index*2 - 1;
48     K(i:i+1, j:j+1) = K_sum;
49     % Symmetric matrix
50     K(j:j+1, i:i+1) = K_sum;
51
52     % Special case of top nodes
53     if mod(elements(index).node_m,5) == 0
54         n1 = nodes(elements(index).node_m);
55         n2 = nodes(elements(index+1).node_m);
56         % There is only one element that touches this line
57         e1 = index+1;
58         e2 = -1;
59         K_sum = getNonDiagonalKSum(n1,n2,e1,e2,elements);
60         i = n1.index*2 - 1;
61         j = n2.index*2 - 1;
62         K(i:i+1, j:j+1) = K_sum;
63         K(j:j+1, i:i+1) = K_sum;
64     end

```

```

65
66 % Vertical lines
67 n1 = nodes(elements(index).node_i);
68 n2 = nodes(elements(index).node_m);
69 [e1, e2] = getTouchingElements(n1,n2);
70 K_sum = getNonDiagonalKSum(n1,n2,e1,e2,elements);
71 i = n1.index*2 - 1;
72 j = n2.index*2 - 1;
73 K(i:i+1, j:j+1) = K_sum;
74 % Symmetric matrix
75 K(j:j+1, i:i+1) = K_sum;
76
77 % Special case of last column
78 if index > 120
79     n1 = nodes(elements(index+1).node_j);
80     n2 = nodes(elements(index+1).node_m);
81     [e1, e2] = getTouchingElements(n1,n2);
82     K_sum = getNonDiagonalKSum(n1,n2,e1,e2,elements);
83     i = n1.index*2 - 1;
84     j = n2.index*2 - 1;
85     K(i:i+1, j:j+1) = K_sum;
86     % Symmetric matrix

```

```

87         K(j:j+1, i:i+1) = K_sum;
88     end
89
90     % Diagonal lines
91     % No special cases.
92     n1 = nodes(elements(index).node_j);
93     n2 = nodes(elements(index).node_m);
94     e1 = index;
95     e2 = index+1;
96     K_sum = getNonDiagonalKSum(n1,n2,e1,e2,elements);
97     i = n1.index*2 - 1;
98     j = n2.index*2 - 1;
99     K(i:i+1, j:j+1) = K_sum;
100     % Symmetric matrix
101     K(j:j+1, i:i+1) = K_sum;
102 end
103
104 clear index i j e e1 e2 n1 n2 K_sum n1 length

```

Helper Functions

These functions are used to clean up `getTotalK.m` and reduce duplication. In retrospect, `getTouchingElements.m` is overkill (in fact, wasted resources), since we always know which elements are touching a line given a certain case. However, there are fewer conditional statements this way.

`getDiagonalKSum.m`

```
1 function K_sum = getDiagonalKSum(node,elements)
2     length = size(node.elements,2);
3     K_sum = [ 0 0 ; 0 0 ];
4     for z=1:length
5         e = node.elements(z);
6         i = (elements(e).index_map(node.index))*2-1;
7         K_sum = K_sum + elements(e).K(i:i+1, i:i+1);
8     end
9 end
```

getTouchingElements.m

```
1 function [e1, e2] = getTouchingElements(n1, n2)
2     % This function finds the two elements that
3     % are both touching node1 and node2 in  $O(N\log N)$  time
4     % (Node.elements is pre-sorted so ismember does binary search)
5     intersection = ismember(n1.elements, n2.elements);
6     s = size(intersection,2);
7     e1 = -1;
8     e2 = -1;
9     for i=1:s
10         if intersection(i) == 1 && e1 == -1
11             e1 = i;
12         elseif intersection(i) == 1
13             e2 = i;
14             break
15         end
16     end
17     e1 = n1.elements(e1);
18     if e2 > 0
19         e2 = n1.elements(e2);
20     end
21 end
```

getNonDiagonalKSum.m

```
1 function K_sum = getNonDiagonalKSum(n1, n2, e1, e2, elements)
2     % This function calculates the K submatrix for two nodes
3     % (i.e. lines)
4     % This involves summing K submatrices of 2 elements if two elements
5     % are touching both nodes (otherwise we return the K submatrix of the
6     % only element that touches the node)
7
8     i1 = (elements(e1).index_map(n1.index))*2-1;
9     j1 = (elements(e1).index_map(n2.index))*2-1;
10    K_sum = elements(e1).K(i1:i1+1, j1:j1+1);
11    if e2 > 0
12        i2 = (elements(e2).index_map(n1.index))*2-1;
13        j2 = (elements(e2).index_map(n2.index))*2-1;
14        K_sum = K_sum + elements(e2).K(i2:i2+1, j2:j2+1);
15    end
16 end
```