Implementation of an isoparametric membrane finite element

Course of Spacecraft Structures

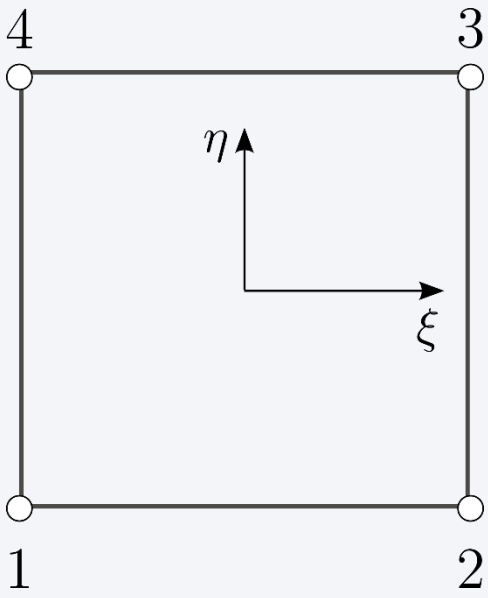
AY 2017/2018

Riccardo Vescovini

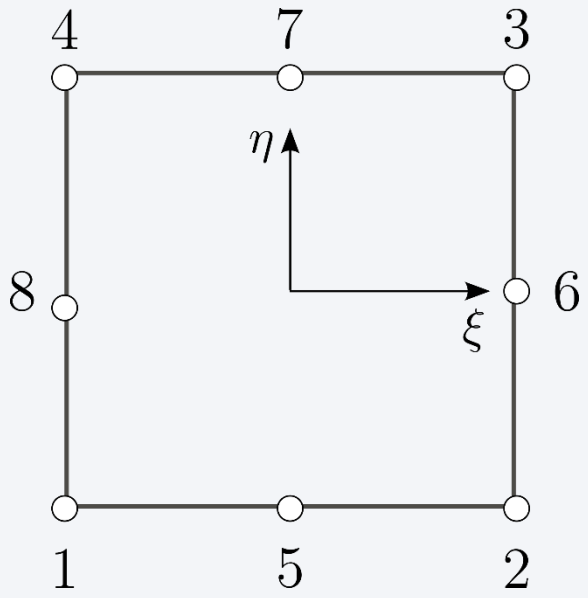
Politecnico di Milano, Department of Aerospace Science and Technology

Introduction – element types

- Two element types are considered: 4– and 8–node elements. The numbering of the nodes is reported in the figure

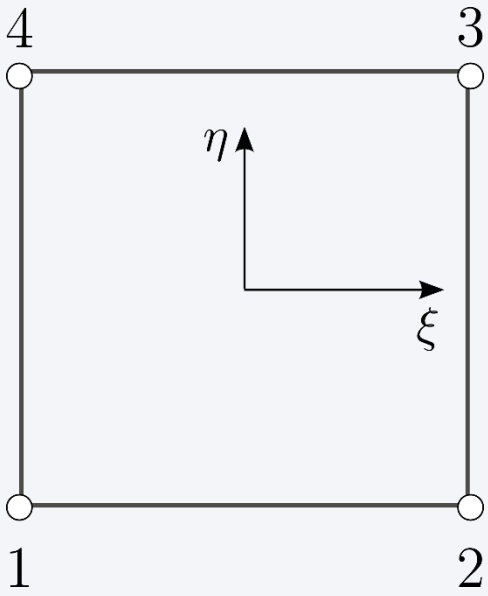


$\xi, \eta \in [-1\,1]$



Introduction – element types

- The shape functions are defined in the computational domain. For the 4–node element they are:



$$N_1 = \frac{1}{4}(1-\xi)(1-\eta)$$

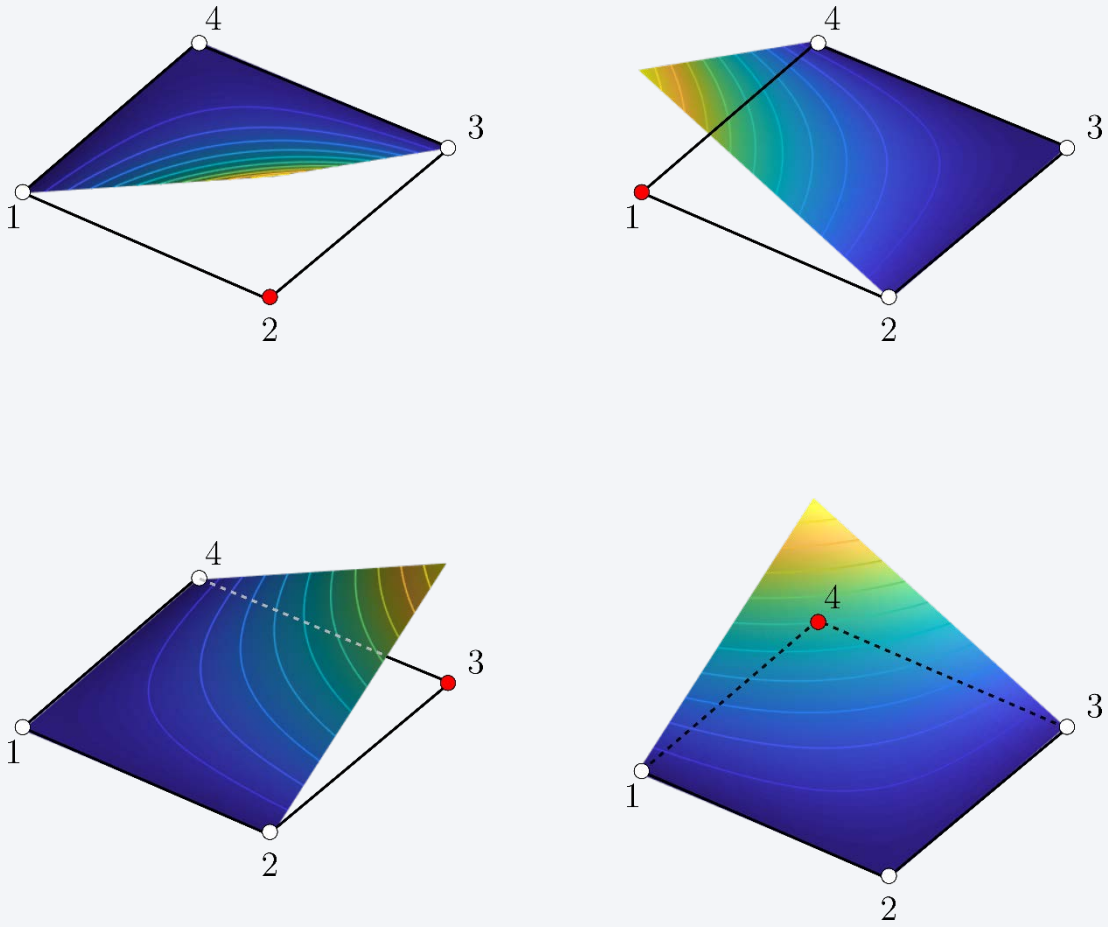
$$N_2 = \frac{1}{4}(1+\xi)(1-\eta)$$

$$N_3 = \frac{1}{4}(1+\xi)(1+\eta)$$

$$N_4 = \frac{1}{4}(1-\xi)(1+\eta)$$

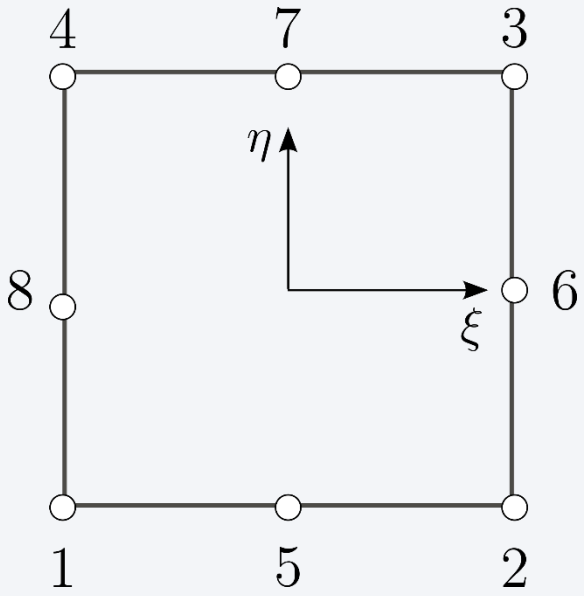
Introduction – element types

- Shape functions of 4-node elements



Introduction – element types

- The shape functions are defined in the computational domain. For the 8–node element they are:



$$N_1 = \frac{1}{4}(1-\xi)(1-\eta)(-\xi-\eta-1)$$

$$N_2 = \frac{1}{4}(1+\xi)(1-\eta)(\xi-\eta-1)$$

$$N_3 = \frac{1}{4}(1+\xi)(1+\eta)(\xi+\eta-1)$$

$$N_4 = \frac{1}{4}(1-\xi)(1+\eta)(-\xi+\eta-1)$$

$$N_5 = \frac{1}{2}(1-\xi^2)(1-\eta)$$

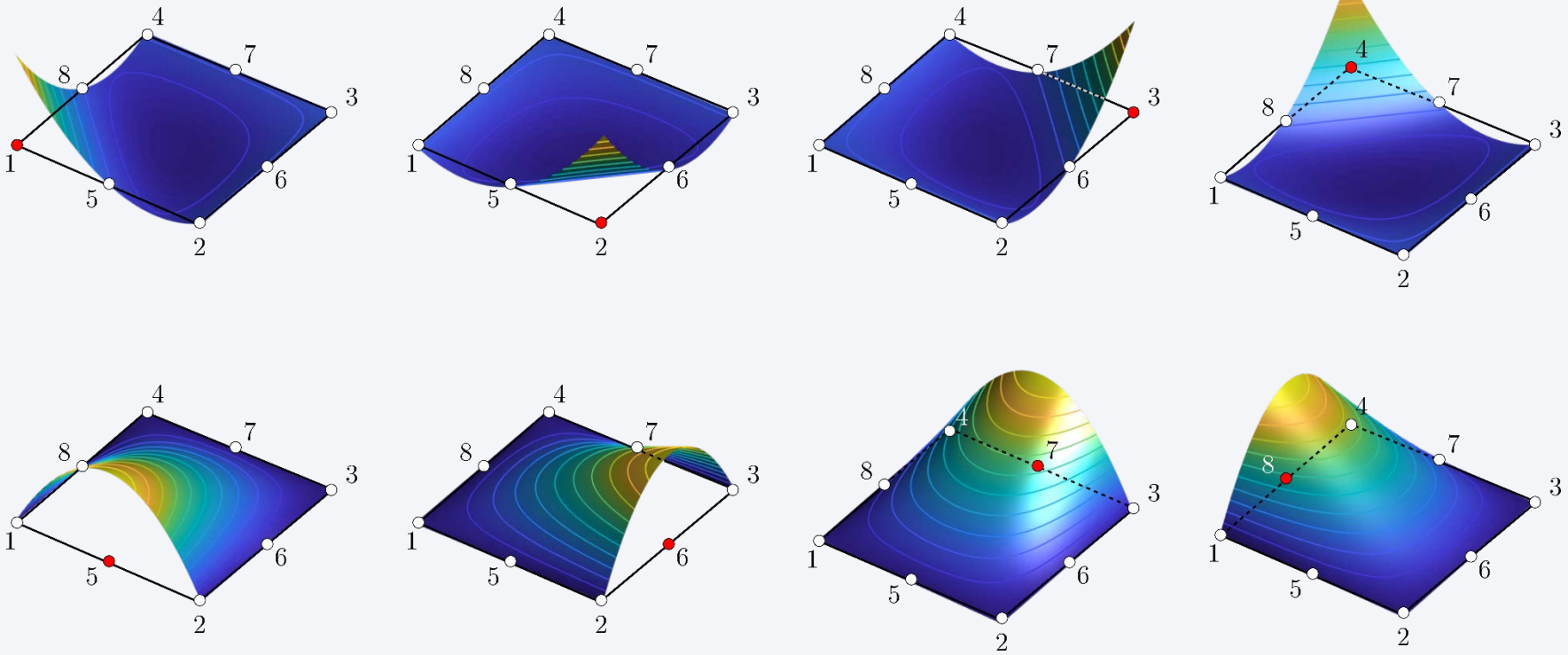
$$N_6 = \frac{1}{2}(1+\xi)(1-\eta^2)$$

$$N_7 = \frac{1}{2}(1-\xi^2)(1+\eta)$$

$$N_8 = \frac{1}{2}(1-\xi)(1-\eta^2)$$

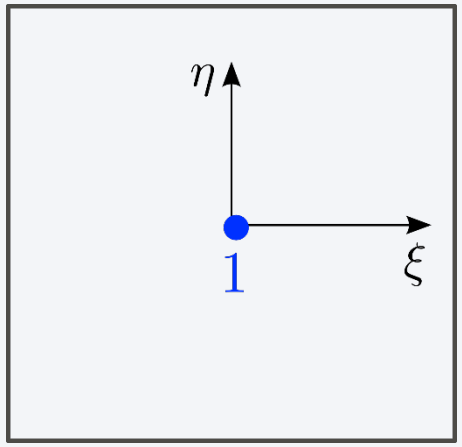
Introduction – element types

- Shape functions of 8-node elements



Introduction – numerical integration

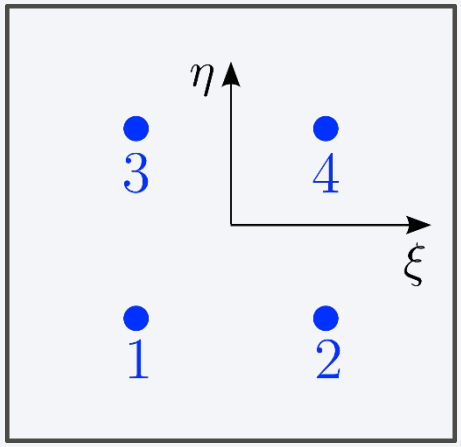
- Gauss integration is adopted for integrating the stiffness matrix. The numbering of the integration points, their position and the corresponding weights are summarized here below up to the 3×3 integration scheme



1×1 rule

$x_i = 0$

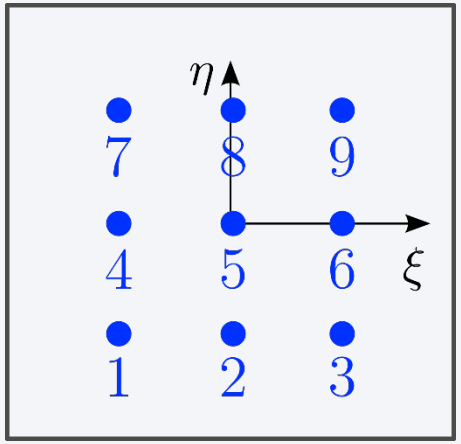
$w_i = 2$



2×2 rule

$x_i = \left\{-1/\sqrt{3} \quad 1/\sqrt{3}\right\}$

$w_i = \{1 \quad 1\}$



3×3 rule

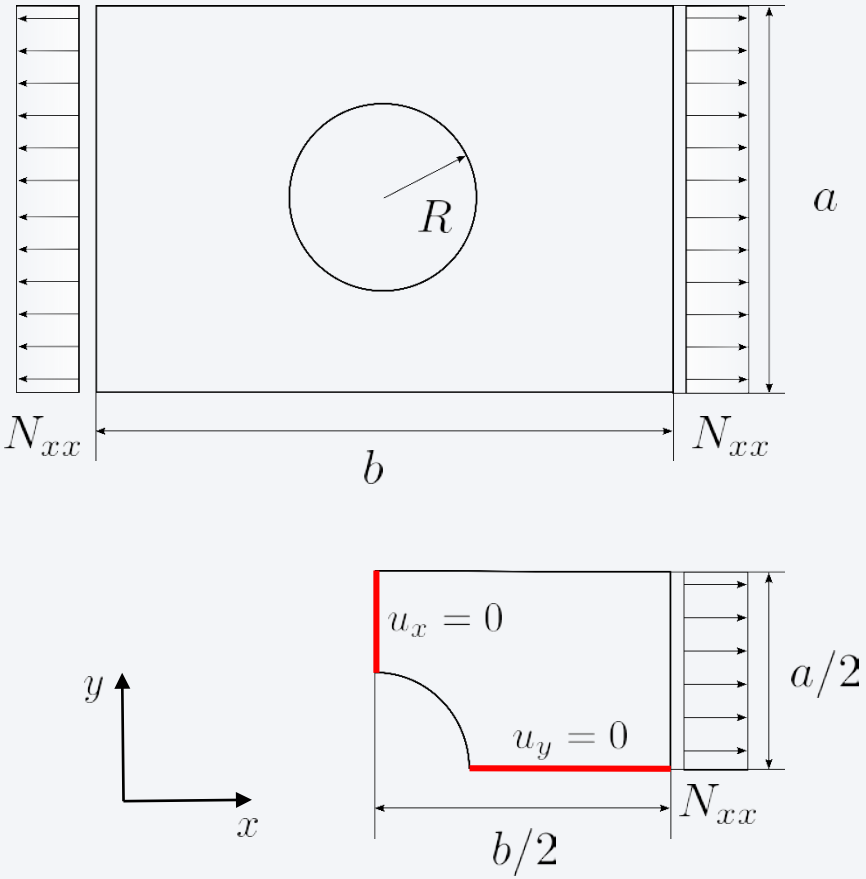
$x_i = \left\{-\sqrt{.6} \quad 0 \quad \sqrt{.6}\right\}$

$w_i = \{5/9 \quad 8/9 \quad 5/9\}$

x_i: Gauss points positions
w_i: Gauss points weights

Example

- Open-hole test: membrane with center hole, loaded in traction (1/4 of the structure due to the double symmetry of the problem)

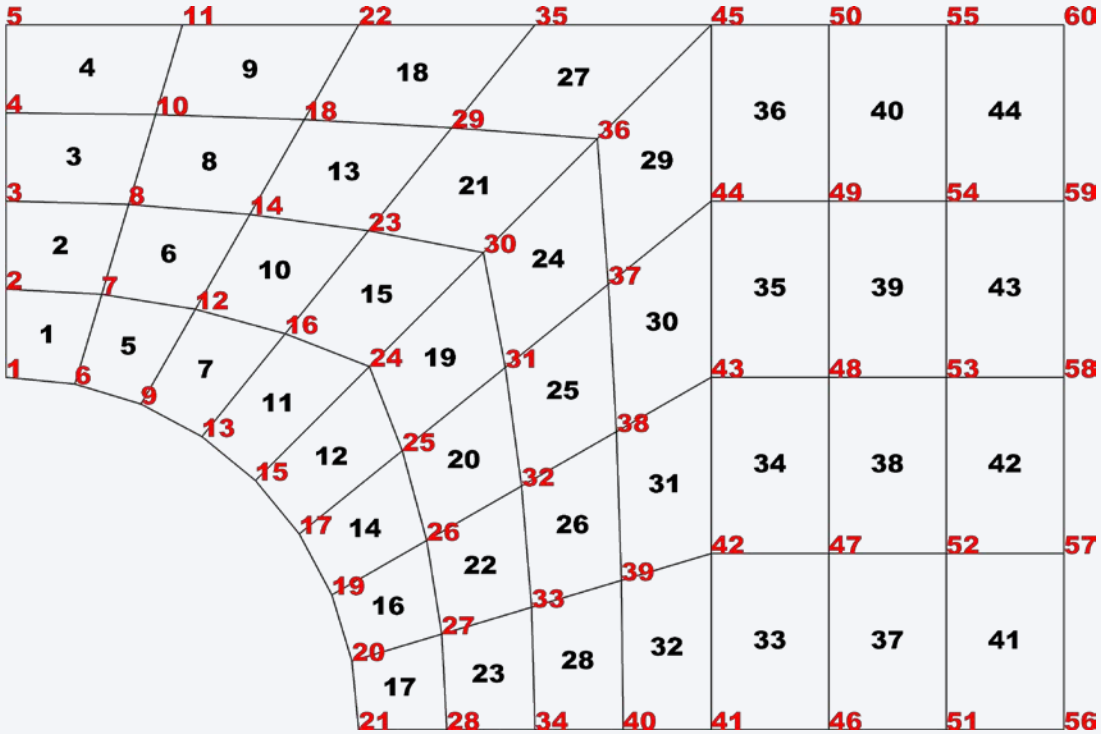


Input data

a = 100 mm
b = 150 mm
R = 25 mm
t = 1 mm
E = 72 GPa
v = 0.3
N_{xx} = 20 N/mm

Example

- The global reference system, the node and element numbering are taken as reported in the figure (these choices are clearly arbitrary)



Red: node IDs

Black: element IDs

Overview of the program – Main

- The structure of the program consists of three distinct steps:
  - the preparation of data (pre-process)
  - the analysis of the structure (solution)
  - analysis of results (post-processing)

- The three steps are accomplished by the three functions `input_open_hole_bilinear_44_els` (example taken as reference), `fem_solver` and `plot_def_vs_undef` (other post-processing functions can be considered)

```
% --- 1. Pre-process
INPUT = input_open_hole_bilinear_44_els;

% --- 2. Solution
[ MODEL, PROPERTY, POST ] = fem_solver( INPUT );

% --- 3. Post-process - plot results
plot_def_vs_undef( MODEL, POST );
```

Overview of the program – Structures

- Three functions are used in in the program MODEL, PROPERTY, POST. Most of data are stored in MODEL, whose fields are

```
MODEL =
  struct with fields:
       elements: [44×4 double]    elements: [ID_node1 ID_node2 ID_node3 ID_node4]
       int_rule: [1×1 struct]     int_rule: structure with integration points and weights
          nodes: [60×2 double]    nodes: [x_coord y_coord]
            pos: [60×2 double]    pos: position of the nodal dofs in the global vector
           ndof: 120              ndof: total number of dofs
           nels: 44               nels: total number of elements
         nnodes: 60               nnodes: total number of nodes
         eltype: 4                eltype: number of nodes per element
              K: [107×107 double] K: stiffness matrix (constrained structure)
              F: [107×1 double]   F: vector of loads (constrained structure)
    constr_dofs: [13×1 double]    constr_dofs: vector of constrained dofs
      free_dofs: [1×107 double]   free_dofs: vector of free dofs
     nfree_dofs: 107              nfree_dofs: free degrees of freedom
           ptrs: [44×8 double]    ptrs: vectors of pointers
          K_unc: [120×120 double] K_unc: stiffness matrix (unconstrained structure)
          F_unc: [120×1 double]   F_unc: vector of loads (unconstrained structure)
              U: [107×1 double]   U: global vector of displacements (constr struct)
          U_unc: [120×1 double]   U_unc: global vector of displacements (unconstr struct)
```

Overview of the program – Structures

- The structure PROPERTY stores the elastic properties of the membrane. Although not strictly necessary as a separate structure, it is here used because useful when extending the analysis to the case of composite structures

```
PROPERTY =

  struct with fields:

    t: 1.00                    t: thickness
    A: [3×3 double]            A: membrane constitutive law
```

Overview of the program – Structures

- The structure of POST collects some data evaluated during the post-processing analysis

```
POST =
  struct with fields:
           sxx_ip: [44×4 double]   sxx_ip: stress sxx at the integration points
           syy_ip: [44×4 double]   syy_ip: stress syy at the integration points
           sxy_ip: [44×4 double]   sxy_ip: stress sxy at the integration points
              sxx: [44×4 double]   sxx_ip: stress sxx at the corners
              syy: [44×4 double]   syy_ip: stress syy at the corners
              sxy: [44×4 double]   sxy_ip: stress sxy at the corners
     sxx_centroid: [44×1 double]   sxx_ip: stress sxx at the centroid
     syy_centroid: [44×1 double]   syy_ip: stress syy at the centroid
     sxy_centroid: [44×1 double]   sxy_ip: stress sxy at the centroid
```

Step 1, pre-process: input_model

- The input file is written from the user by specifying the characteristics of the model to be analyzed. The data are organized into the structure INPUT, which is divided into different fields

```
function INPUT = input_open_hole_bilinear_44_els


INPUT =

  struct with fields:

    integration_pts: 2
           elements: [44×4 double]
              nodes: [60×2 double]
                  E: 72000
                 nu: 0.3000
                  t: 1.0000
               load: [5×3 double]
                spc: [13×2 double]
```

```
integration_pts: number of integration pts
       elements: [ID_node1 .... ID_node4]
          nodes: [x_coord y_coord]
              E: Young's moduls
             nu: Poisson's ratio
              t: thickness
           load: [ID_node dof magnitude]
            spc: [ID_node constrained_dof]
```

Step 1, pre-process: input_open_hole_bilinear_44_els

```
function INPUT = input_open_hole_bilinear_44_els

% -- Init
INPUT = struct();

% -- Elements
INPUT.integration_pts = 2;

INPUT.elements = [  1      6      7      2
 2      7      8      3
 3      8     10      4
.
.
.
54     59     60     55 ];

% -- Nodes
INPUT.nodes = [ 0.         25.
       0.       31.25
.
.
      75.         50.  ];
```

```
% -- Section properties
INPUT.E = 72000;
INPUT.nu = 0.3;
INPUT.t = 1.0;

% -- Loading conditions
INPUT.load = [   56  1 125.
 57  1 250.
 58  1 250.
 59  1 250.
 60  1 125. ];

% -- Boundary conditions
INPUT.spc = [  1    1
 2    1
 3    1
.
.
.
51    2
56    2 ];
```

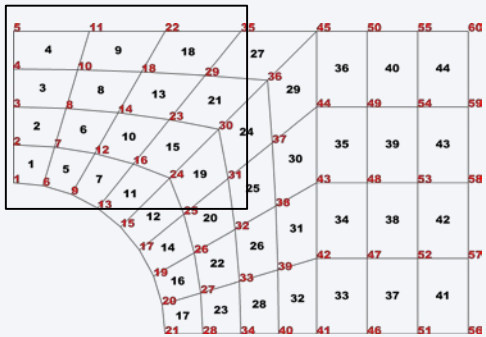
Step 1, pre-process: `input_model`

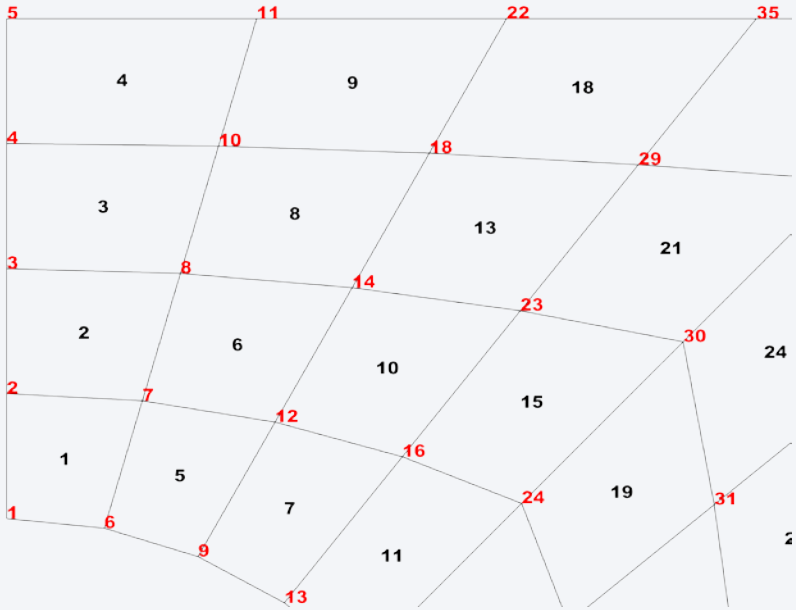
- With reference to the present example:

```
INPUT.elements

ans =

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

Element 1: nodes 1, 6, 7 and 2

Element 2: nodes 2, 7, 8 and 3

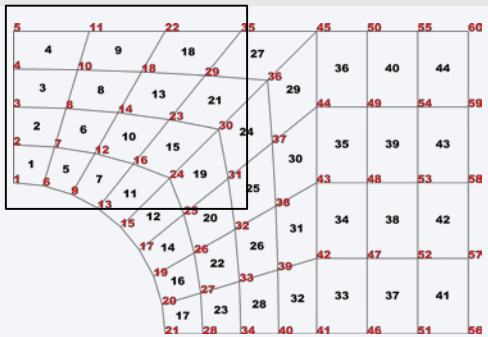


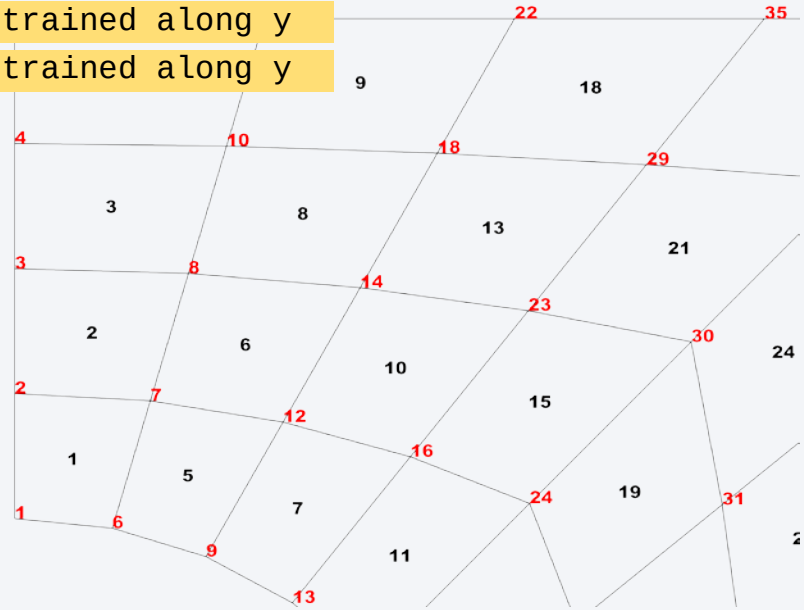


Step 1, pre-process: input_model

- With reference to the present example:

```
INPUT.spc

ans =

     1     1   ──────────►  Node 1, constrained along x
     2     1   ──────────►  Node 2, constrained along x
     3     1
     .
     .
    51     2   ──────────►  Node 51, constrained along y
    56     2   ──────────►  Node 56, constrained along y
```





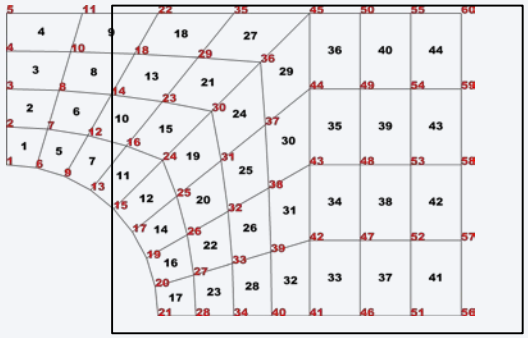
Step 1, pre-process: `input_model`

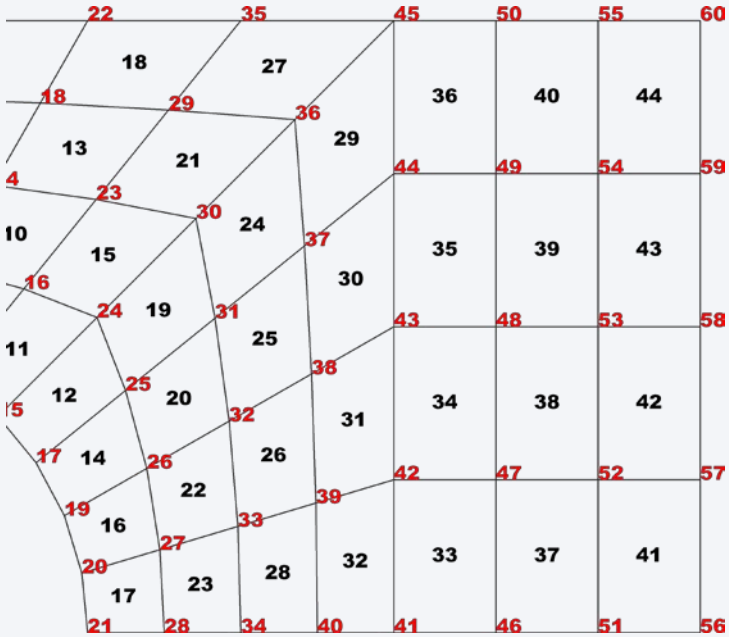
- With reference to the present example:

```
INPUT.load

ans =

    56     1   125
    57     1   250
    58     1   250
    59     1   250
    60     1  1252
```

Node 56, loaded along x, force magnitude 125

Node 57, loaded along x, force magnitude 250





Step 2, solution: fem_solver

- The function fem_solver represents the core of the program, and is divided into a number of functions

```
function [ MODEL, PROPERTY, POST ] = fem_solver( INPUT )

% --- Set model
[ MODEL, PROPERTY, POST ] = set_model( INPUT );

% --- Set pointers
MODEL = set_pointers( MODEL );

% --- Build & assembly matrices
MODEL = build_K_matrix( MODEL, PROPERTY );

% --- Impose constraints and solve
MODEL = solve_structure( MODEL );

% --- Stress recovery
POST = stress_recovery( MODEL, PROPERTY, POST );
```

Step 2, solution: fem_solver

```
function [ MODEL, PROPERTY, POST ] = fem_solver( INPUT )

% --- Set model
[ MODEL, PROPERTY, POST ] = set_model( INPUT );

% --- Set pointers
MODEL = set_pointers( MODEL );

% --- Build & assembly matrices
MODEL = build_K_matrix( MODEL, PROPERTY );

% --- Impose constraints and solve
MODEL = solve_structure( MODEL );

% --- Stress recovery
POST = stress_recovery( MODEL, PROPERTY, POST );
```

Step 2, solution: `set_model`

- The MODEL structure is returned as:

```
MODEL =
  struct with fields:
       elements: [44×4 double]
       int_rule: [1×1 struct]
          nodes: [60×2 double]
            pos: [60×2 double]
           ndof: 120
           nels: 44
         nnodes: 60
         eltype: 4
              K: [107×107 double]
              F: [107×1 double]
    constr_dofs: [13×1 double]
      free_dofs: [1×107 double]
     nfree_dofs: 107
           ptrs: (filled later)
          K_unc: (filled later)
          F_unc: (filled later)
              U: (filled later)
          U_unc: (filled later)
```

Step 2, solution: `set_model`

- where:

```
MODEL.elements
ans =
     1     6     7     2
     2     7     8     3
     3     8    10     4
     4    10    11     5


MODEL.nodes
ans =
         0   25.0000
         0   31.2500
         0   37.5000
         0   43.7500
         0   50.0000
    4.8773   24.5196
```

Element 1: nodes 1, 6, 7 and 2

Element 2: nodes 2, 7, 8 and 3

Node 1: x = 0, y = 25

Node 2: x = 0, y = 31.25

Step 2, solution: `set_model`

- where:

```
MODEL.int_rule
ans =
  struct with fields:
    x: [2×1 double]
    w: [2×1 double]
```

→ Gauss point position

→ Gauss point weights

```
% --- 1 point
x = [ 0 ];
w = [ 2 ];

% --- 2 points
x = [ -1/sqrt(3) 1/sqrt(3) ];
w = [ 1 1 ];

% --- 3 points
x = [ -sqrt(.6) 0 sqrt(.6) ];
w = [ 5/9 8/9 5/9 ];
```

```
MODEL.pos
ans =
     1    61
     2    62
     3    63
     .
     .
     .
     7    67
    59   119
    60   120
```

→ for node 59, Ux is in position 59, Uy position 119

→ for node 60, Ux is in position 60, Uy position 120

DOFs are sorted as:

$$\mathbf{U}^{\mathrm{T}} = \begin{bmatrix} U_x^1 & U_x^2 & \dots & U_x^N & U_y^1 & U_2^y & \dots & U_y^N \end{bmatrix}$$

Step 2, solution: `set_model`

- and:

```
PROPERTY =
  struct with fields:
    t: 1.6000
    A: [3×3 double]
```

`thickness`

membrane stiffness

$$\mathbf{A} = \frac{Et}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$$

```
POST =
  struct with fields:
            sxx_ip: (filled later)
            syy_ip: (filled later)
            sxy_ip: (filled later)
               sxx: (filled later)
               syy: (filled later)
               sxy: (filled later)
      sxx_centroid: (filled later)
      syy_centroid: (filled later)
      sxy_centroid: (filled later)
```

Step 2, solution: fem_solver

```
function [ MODEL, PROPERTY, POST ] = fem_solver( INPUT )

% --- Set model
[ MODEL, PROPERTY, POST ] = set_model( INPUT );

% --- Set pointers
MODEL = set_pointers( MODEL );

% --- Build & assembly matrices
MODEL = build_K_matrix( MODEL, PROPERTY );

% --- Impose constraints and solve
MODEL = solve_structure( MODEL );

% --- Stress recovery
POST = stress_recovery( MODEL, PROPERTY, POST );
```

Step 2, solution: `set_pointers`

- The function writes the pointers associated with the generic i-th element in the field MODEL(i).ptrs

```
function MODEL = set_pointers( MODEL )

MODEL.ptrs

ans =

     1     6     7     2    61    66    67    62
     2     7     8     3    62    67    68    63
     3     8    10     4    63    68    70    64
     4    10    11     5    64    70    71    65
```

→ Position of dofs of element 1

→ Position of dofs of element 2

Step 2, solution: fem_solver

```
function [ MODEL, PROPERTY, POST ] = fem_solver( INPUT )

% --- Set model
[ MODEL, PROPERTY, POST ] = set_model( INPUT );

% --- Set pointers
MODEL = set_pointers( MODEL );

% --- Build & assembly matrices
MODEL = build_K_matrix( MODEL, PROPERTY );

% --- Impose constraints and solve
MODEL = solve_structure( MODEL );

% --- Stress recovery
POST = stress_recovery( MODEL, PROPERTY, POST );
```

Step 2, solution: equations reminder

- Jacobian of the transformation

$$\mathbf{J} = \begin{bmatrix} N_{1,\xi} & N_{2,\xi} & \dots & N_{N,\xi} \\ N_{1,\eta} & N_{2,\eta} & \dots & N_{N,\eta} \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_N & y_N \end{bmatrix}$$

N: number of nodes
(4: bilinear element;
8: biquadratic element)

```
Derivatives in the
computational domain;
available by the
definitions of N_i
```

```
Coordinates
of the nodes
composing
the element
```

- Derivatives of the shape functions with respect to x and y

$$\begin{bmatrix} N_{1,x} & N_{2,x} & \dots & N_{N,x} \\ N_{1,y} & N_{2,y} & \dots & N_{N,y} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} N_{1,\xi} & N_{2,\xi} & \dots & N_{N,\xi} \\ N_{1,\eta} & N_{2,\eta} & \dots & N_{N,\eta} \end{bmatrix}$$

Step 2, solution: equations reminder

- B-matrix

$$\mathbf{B} = \begin{bmatrix} N_{1,x} & \dots & N_{N,x} & 0 & \dots & 0 \\ 0 & \dots & 0 & N_{1,y} & \dots & N_{N,y} \\ N_{1,y} & \dots & N_{N,y} & N_{1,x} & \dots & N_{N,x} \end{bmatrix}$$

Obtained by properly organizing the results obtained in the previous step

- Stiffness matrix

$$\mathbf{K} = \int_{-1}^{1}\int_{-1}^{1} \mathbf{B}^{\mathrm{T}}\mathbf{A}\mathbf{B}\det\mathbf{J}\mathrm{d}\xi\,\mathrm{d}\eta = \\ = \sum_{i=1}^{N_{ip}}\sum_{j=1}^{N_{ip}} w_{ij}\mathbf{B}^{\mathrm{T}}(\xi_i,\eta_j)\mathbf{A}\mathbf{B}(\xi_i,\eta_j)\det\mathbf{J}$$

with: $\mathbf{A} = \mathbf{C}t = \dfrac{Et}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$

Step 2, solution: build_K_matrix (1/2)

```
function MODEL = build_K_matrix( MODEL, PROPERTY )

A = PROPERTY.A;

% --- Integration rule parameters
xGauss = MODEL.int_rule.x;
wGauss = MODEL.int_rule.w;
nint_p = length( xGauss );

for i = 1 : MODEL.nels

    % --- Initialize
    K_el = zeros( 2 * MODEL.eltype, 2 * MODEL.eltype );

    % --- Matrix of nodal coordinates
    el_nodes = MODEL.elements( i, : );
    xy_nodes = [ MODEL.nodes( el_nodes, 1 ) MODEL.nodes( el_nodes, 2 ) ];
```

Loop over the elements

xy coordinates of the nodes composing the element (needed to build the B matrix)

```
%  xy_nodes = [ x1 y1;
%               x2 y2;
%               x3 y3;
%               x4 y4 ]; % (in case of 4 nodes)
```

Step 2, solution: build_K_matrix (2/2)

```
    % --- Integrate stiffness matrix
    for iG = 1 : nint_p
        for jG = 1 : nint_p

            xi  = xGauss( jG );
            eta = xGauss( iG );

            [ B, detJ ] = get_B_matrix( xi, eta, xy_nodes );

            % --- Stiffness matrix
            K_el = K_el + wGauss( iG ) * wGauss( jG ) * ( B' * A * B * detJ );

        end
    end

    % --- Assemble contribution
    ptrs = MODEL.ptrs( i, : );
    MODEL.K( ptrs, ptrs ) = MODEL.K( ptrs, ptrs ) + K_el;

end
```

Loop over x- and y-direction for numerical integration

Build matrix B and the Jacobian of the isoparametric transformation

Step 2, solution: fem_solver

```
function [ MODEL, PROPERTY, POST ] = fem_solver( INPUT )

% --- Set model
[ MODEL, PROPERTY, POST ] = set_model( INPUT );

% --- Set pointers
MODEL = set_pointers( MODEL );

% --- Build & assembly matrices
MODEL = build_K_matrix( MODEL, PROPERTY );

% --- Impose constraints and solve
MODEL = solve_structure( MODEL );

% --- Stress recovery
POST = stress_recovery( MODEL, PROPERTY, POST );
```

→ Just remove rows and columns

Step 2, solution: fem_solver

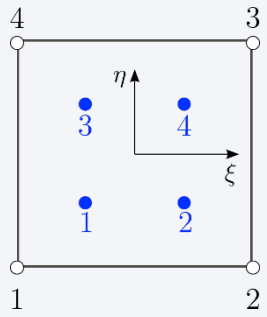
```
function [ MODEL, PROPERTY, POST ] = fem_solver( INPUT )

% --- Set model
[ MODEL, PROPERTY, POST ] = set_model( INPUT );

% --- Set pointers
MODEL = set_pointers( MODEL );

% --- Build & assembly matrices
MODEL = build_K_matrix( MODEL, PROPERTY );

% --- Impose constraints and solve
MODEL = solve_structure( MODEL );

% --- Stress recovery
POST = stress_recovery( MODEL, PROPERTY, POST );
```

Step 2, solution: `stress_recovery`

- Once the nodal displacements are available, it is possible to determine the strains and the stresses in any desired position. Generally, a good strategy consists in evaluating the stresses at the integration points (another common approach consists in evaluating the stresses at the "Barlow points", viz. the integration points corresponding to an integration scheme with N-1 points)

- Stresses at the integration points

$$\begin{Bmatrix} \sigma_{xx}\left(\xi_i, \eta_j\right) \\ \sigma_{yy}\left(\xi_i, \eta_j\right) \\ \sigma_{xy}\left(\xi_i, \eta_j\right) \end{Bmatrix} = \mathbf{CB}\left(\xi_i, \eta_j\right) \mathbf{U}$$



(B matrix evaluated at the integration points)

- In general, it can useful to evaluate the stresses in correspondence of the nodes. Indeed, the integration points are inside the element, while the maximum levels of stresses can be reached on the outer positions (consider, for instance, the planar bending of a beam). The direct evaluation of the stresses is not a good choice, as poor quality results are obtained

Step 2, solution: `stress_recovery`

- A common strategy consists in adopting an interpolation scheme, where a polynomial expression, consistent with the order of the integration rule adopted, is used.
- For a 2×2 integration rule, the stresses are available at four integration points. The interpolation scheme is then taken in the form:

$$\sigma_{ik}(\xi,\eta) = a_0 + a_1\xi + a_2\eta + a_3\xi\eta$$

- and the constants a_i are obtained by solving the linear system obtained by imposing:

$$\begin{cases} a_0 + a_1\xi_1 + a_2\eta_1 + a_3\xi_1\eta_1 = \sigma_{ik}(\xi_1,\eta_1) \longrightarrow \texttt{integration point 1} \\ \vdots \\ a_0 + a_1\xi_4 + a_2\eta_4 + a_3\xi_4\eta_4 = \sigma_{ik}(\xi_4,\eta_4) \longrightarrow \texttt{integration point 4} \end{cases}$$

- In a similar manner, for a 3×3 integration rule, the interpolation scheme can be taken as:

$$\sigma_{ik}(\xi,\eta) = a_0 + a_1\xi + a_2\eta + a_3\xi\eta + a_4\xi^2 + a_5\eta^2 + a_6\xi^2\eta + a_7\xi\eta^2 + a_8\xi^2\eta^2$$

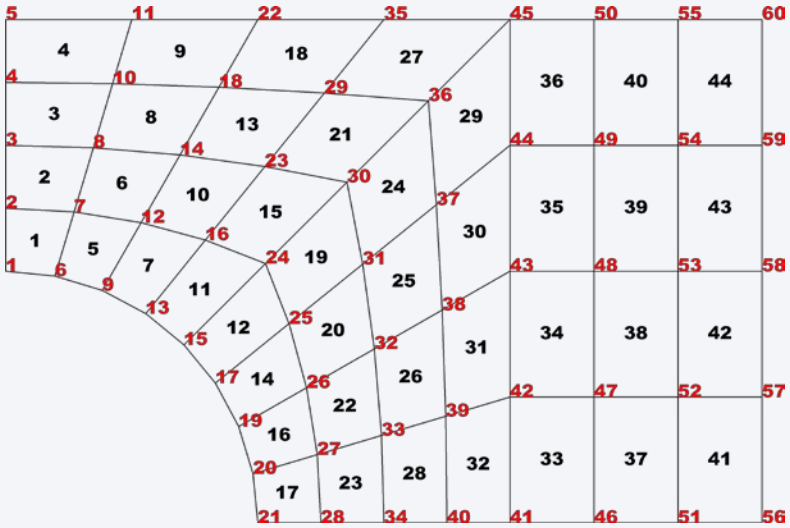
and the linear system to be solved has dimension 9

Exercise

- Complete the program by writing the missing parts of the code; consider both 4- and 8-node elements
- Analyze the problems reported next to compare the correctness of your implementation

Results

- Jacobians



$\mathbf{J}^i(\xi_j, \eta_j)$: Jacobian of the element i at the integration point j

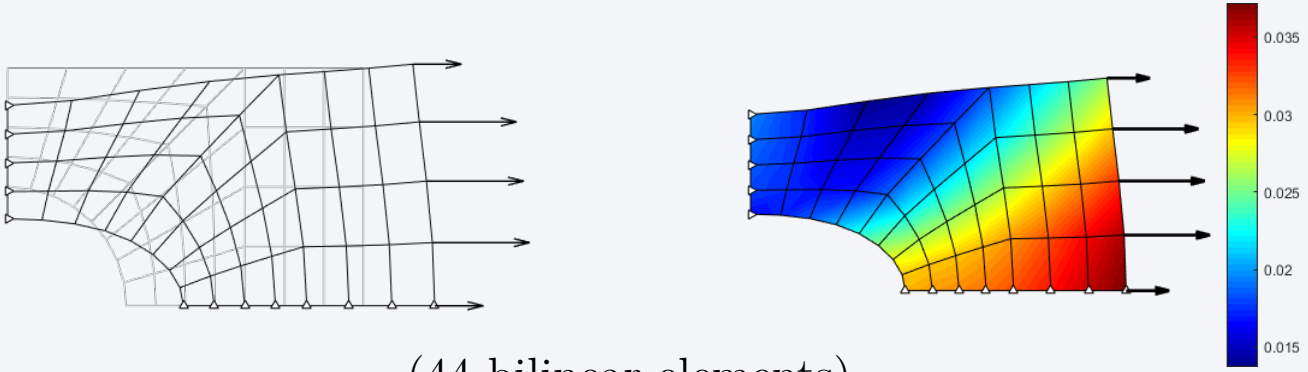
The ratio between the elements of the Jacobian matrix provides a measure of the geometric deformation of the element. In particular, the diagonal terms furnish a measure of the geometric «stretching» of the elements, while the extra-diagonal elements provides a measure of the distortion

$$\mathbf{J}^1(\xi_1, \eta_1) = \begin{bmatrix} 2.6400 & -0.2275 \\ 0.2014 & 3.1377 \end{bmatrix} \longrightarrow \texttt{element 1 stretched and slightly distorted}$$

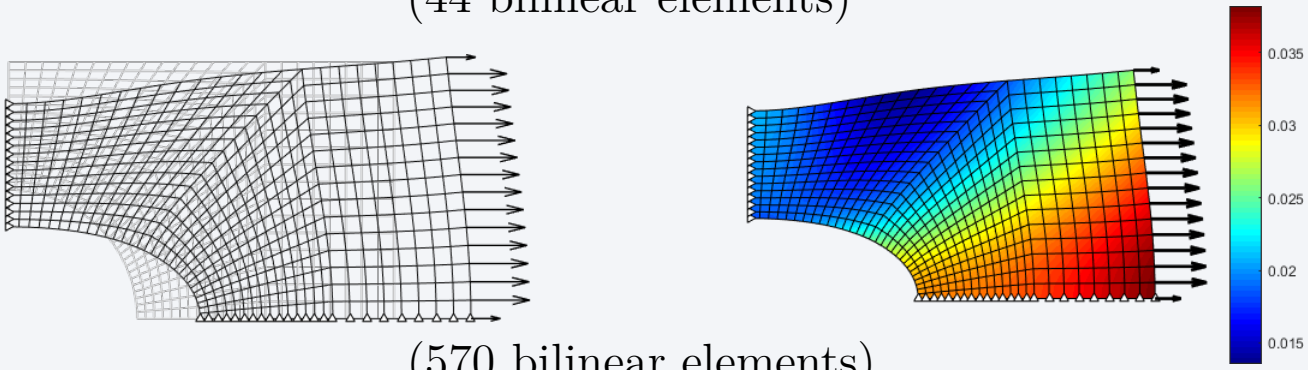
$$\mathbf{J}^{27}(\xi_1, \eta_1) = \begin{bmatrix} 5.3912 & -0.3065 \\ 3.1815 & 3.7338 \end{bmatrix} \longrightarrow \texttt{element 27 stretched and relatively distorted}$$

$$\mathbf{J}^{33}(\xi_1, \eta_1) = \begin{bmatrix} 4.1667 & 0 \\ 0.0000 & 6.2500 \end{bmatrix} \longrightarrow \texttt{element 33 stretched but not distorted}$$

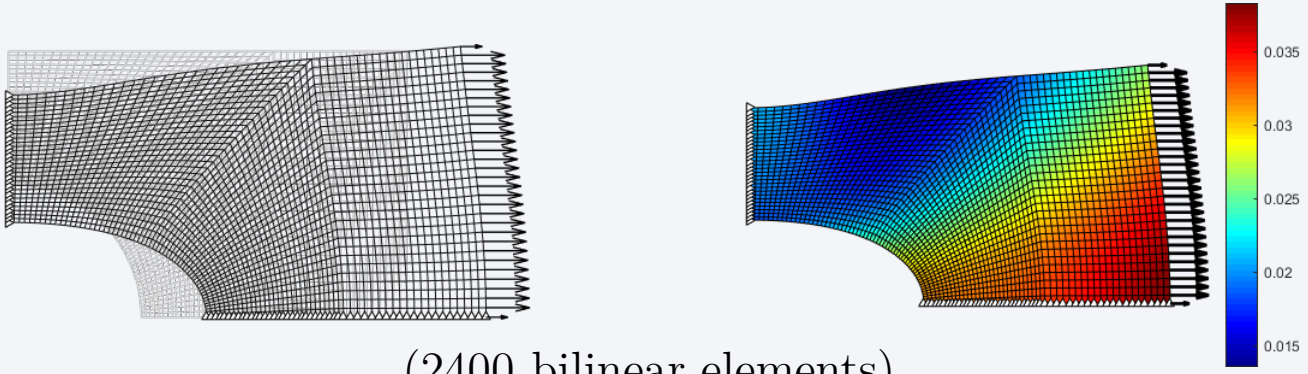
Results: deformed shape and displacement contour



(44 bilinear elements)

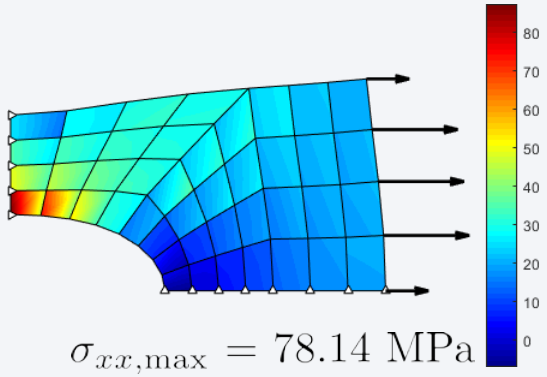


(570 bilinear elements)

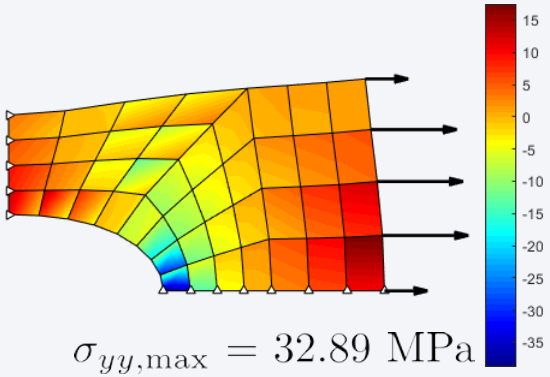


(2400 bilinear elements)

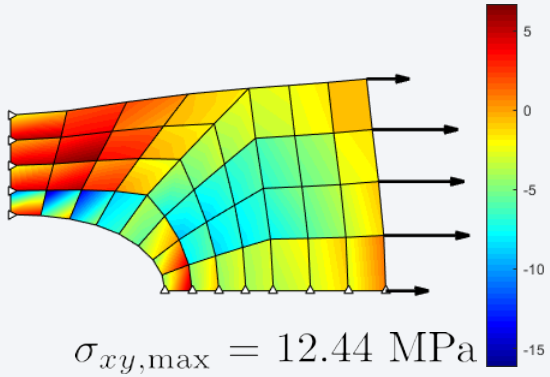
Results: stress components contour



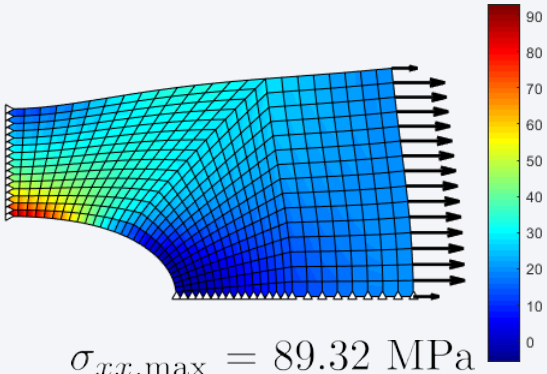
$\sigma_{xx,\max} = 78.14$ MPa



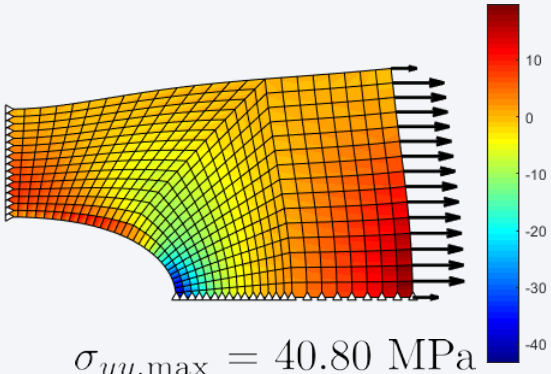
$\sigma_{yy,\max} = 32.89$ MPa



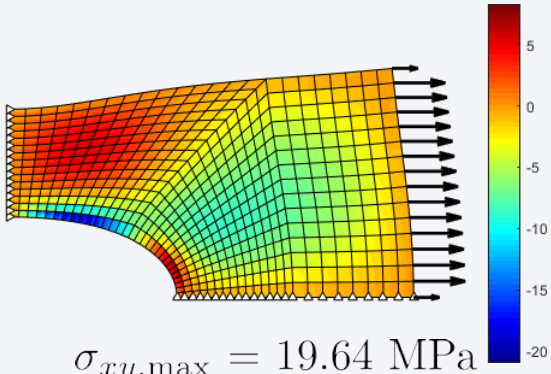
$\sigma_{xy,\max} = 12.44$ MPa



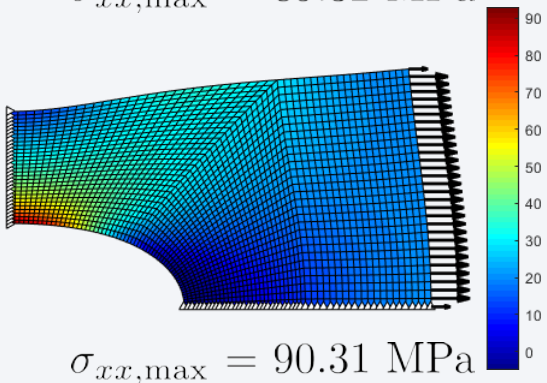
$\sigma_{xx,\max} = 89.32$ MPa



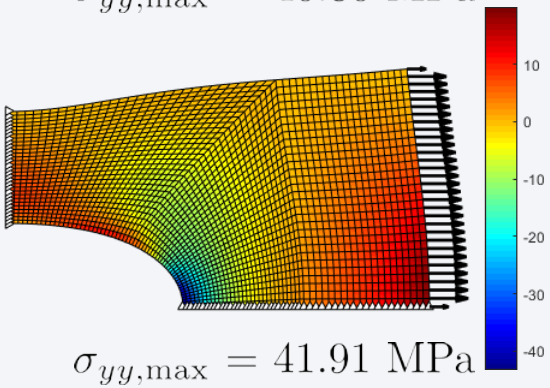
$\sigma_{yy,\max} = 40.80$ MPa



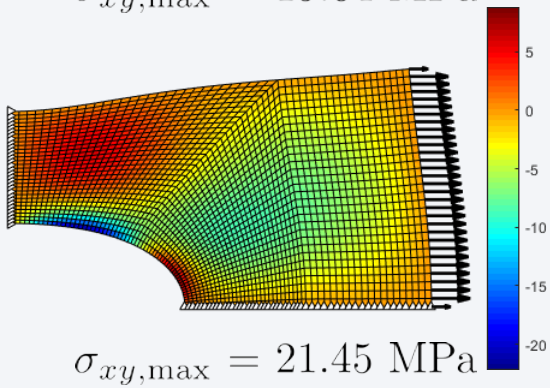
$\sigma_{xy,\max} = 19.64$ MPa



$\sigma_{xx,\max} = 90.31$ MPa



$\sigma_{yy,\max} = 41.91$ MPa



$\sigma_{xy,\max} = 21.45$ MPa

Results: a few comments

- Summary of results: displacements, stresses, energies

	4-nodes			8-nodes		
N_{els}	44	570	2400	44	570	2400
$U_{x,max}$ (mm)	0.0372	0.0382	0.0383	0.0381	0.0383	0.0383
$U_{y,max}$ (mm)	0.0195	0.0209	0.0210	0.0208	0.0210	0.0210
$\lvert\sigma_{xx,max}\rvert$ (MPa)	78.1374	89.3160	90.9138	86.2614	90.4516	91.3662
$\lvert\sigma_{yy,max}\rvert$ (MPa)	32.8878	40.8042	41.9088	38.8173	41.6473	42.2574
$\lvert\sigma_{xy,max}\rvert$ (MPa)	12.4434	19.6433	21.4515	17.7873	21.9655	22.7565
U $(N\,mm)$	16.1996	16.5499	16.5738	16.5151	16.5776	16.5807
$\Pi = U + V$ $(N\,mm)$	-16.1996	-16.5499	-16.5738	-16.5151	-16.5776	-16.5807

- The convergence of the solution can be observed both in terms of displacements and stress components. The quantities are referred to one single point, thus provide information on the local behaviour
- The displacements tend to be underestimated. In this example, coarser meshes are associated with smaller displacements with respect to finer ones. Indeed a coarse mesh is stiffer in comparison to a finer one (note that, however, this is true at global level only, and the energy is the right quantity to be analyzed. It may happen that, at local level, a larger displacement is obtained on a coarse mesh)

Results: a few comments

- Summary of results: displacements, stresses, energies

	4-nodes			8-nodes		
N_{els}	44	570	2400	44	570	2400
$U_{x,max}$ (mm)	0.0372	0.0382	0.0383	0.0381	0.0383	0.0383
$U_{y,max}$ (mm)	0.0195	0.0209	0.0210	0.0208	0.0210	0.0210
$\lvert\sigma_{xx,max}\rvert$ (MPa)	78.1374	89.3160	90.9138	86.2614	90.4516	91.3662
$\lvert\sigma_{yy,max}\rvert$ (MPa)	32.8878	40.8042	41.9088	38.8173	41.6473	42.2574
$\lvert\sigma_{xy,max}\rvert$ (MPa)	12.4434	19.6433	21.4515	17.7873	21.9655	22.7565
U $(N\,mm)$	16.1996	16.5499	16.5738	16.5151	16.5776	16.5807
$\Pi = U + V$ $(N\,mm)$	-16.1996	-16.5499	-16.5738	-16.5151	-16.5776	-16.5807

- Similarly, the stresses tend to be underestimated. This is an important aspect to take into account during the analysis phase, as unconservative results can be obtained due to an excessively coarse mesh
- The results illustrate also the different rate of convergence between displacements and stresses. While a 4-node mesh with 570 elements provides a good estimate of the displacement field, a higher refinement level is needed to properly capture the stress field

Results: a few comments

- Summary of results: displacements, stresses, energies

	4-nodes			8-nodes		
N_{els}	44	570	2400	44	570	2400
$U_{x,max}$ (mm)	0.0372	0.0382	0.0383	0.0381	0.0383	0.0383
$U_{y,max}$ (mm)	0.0195	0.0209	0.0210	0.0208	0.0210	0.0210
$\lvert\sigma_{xx,max}\rvert$ (MPa)	78.1374	89.3160	90.9138	86.2614	90.4516	91.3662
$\lvert\sigma_{yy,max}\rvert$ (MPa)	32.8878	40.8042	41.9088	38.8173	41.6473	42.2574
$\lvert\sigma_{xy,max}\rvert$ (MPa)	12.4434	19.6433	21.4515	17.7873	21.9655	22.7565
U $(N\,mm)$	16.1996	16.5499	16.5738	16.5151	16.5776	16.5807
$\Pi = U + V$ $(N\,mm)$	-16.1996	-16.5499	-16.5738	-16.5151	-16.5776	-16.5807

- The quality of the prediction can be improved by reducing the size of the mesh (h-refinement), or by increasing the order of the interpolating scheme (p-refinement). The comparison between the results obtained with equal number of elements, but different element order, illustrates this latter aspect

Results: a few comments

- Summary of results: displacements, stresses, energies

	4-nodes			8-nodes		
N_{els}	44	570	2400	44	570	2400
$U_{x,max}$ (mm)	0.0372	0.0382	0.0383	0.0381	0.0383	0.0383
$U_{y,max}$ (mm)	0.0195	0.0209	0.0210	0.0208	0.0210	0.0210
$\lvert\sigma_{xx,max}\rvert$ (MPa)	78.1374	89.3160	90.9138	86.2614	90.4516	91.3662
$\lvert\sigma_{yy,max}\rvert$ (MPa)	32.8878	40.8042	41.9088	38.8173	41.6473	42.2574
$\lvert\sigma_{xy,max}\rvert$ (MPa)	12.4434	19.6433	21.4515	17.7873	21.9655	22.7565
U $(N\,mm)$	16.1996	16.5499	16.5738	16.5151	16.5776	16.5807
$\Pi = U + V$ $(N\,mm)$	-16.1996	-16.5499	-16.5738	-16.5151	-16.5776	-16.5807

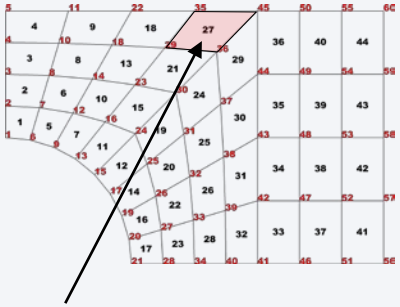
- The strain energy U (or, equivalently, the total potential energy) provides a good measure of the convergence at global level. Note that the strain energy increases as the refinement of the mesh is increased. Indeed, a less stiff model is capable of storing a higher amount of deformation energy. The same idea can be seen the other way around: the total potential energy diminishes as the refinement is increased. Indeed, the minimization of the total potential energy leads to smaller minima as the number of degrees of freedom is increased

Results: a few comments

- Numerical (Gauss) integration, some remarks:
  - an integration of order P allows to integrate exactly a polynomial expressions of order 2P-1. For instance, a cubic expression can be integrated exactly by taking P=2
  - If the element is regular (no distortion), the stiffness matrix is quadratic (in case of 4-node elements). A rule of order 2 sufficies for integrating exactly. This is true if and only if the elements are not distorted. Indeed, the effect of distortion is to transform the expression of the stiffness contribution into a more complex function via the Jacobian of the transformation. In the presence of distorted elements, an integration of higher order than 2 may be needed even for bilinear elements

Results: numerical integration

- Stiffness matrix using different integration schemes



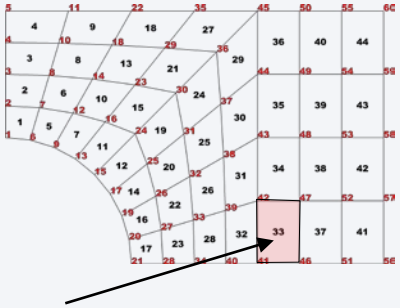
element 27

```
* --- Using 2 x 2 points
K_el =
   1.0e+04 *
    2.6858   -1.6702   -0.6986   -0.3170    0.2167    0.9910   -0.7376   -0.4701
   -1.6702    4.6628   -0.1259   -2.8668    1.1888   -2.1471   -0.7180    1.6763
   -0.6986   -0.1259    2.2720   -1.4475   -0.7376   -0.5202    0.3252    0.9326
   -0.3170   -2.8668   -1.4475    4.6313   -0.6679    1.6763    1.1304   -2.1388
    0.2167    1.1888   -0.7376   -0.6679    2.7418   -0.3144    0.5108   -2.9381
    0.9910   -2.1471   -0.5202    1.6763   -0.3144    9.2936   -2.6253   -6.3540
   -0.7376   -0.7180    0.3252    1.1304    0.5108   -2.6253    2.0645    0.0500
   -0.4701    1.6763    0.9326   -2.1388   -2.9381   -6.3540    0.0500    9.2421

* --- Using 3 x 3 points
K_el =
   1.0e+04 *
    2.6868   -1.6711   -0.6978   -0.3179    0.2164    0.9913   -0.7378   -0.4698
   -1.6711    4.6637   -0.1266   -2.8660    1.1891   -2.1473   -0.7178    1.6760
   -0.6978   -0.1266    2.2726   -1.4482   -0.7378   -0.5200    0.3250    0.9328
   -0.3179   -2.8660   -1.4482    4.6321   -0.6676    1.6760    1.1306   -2.1391
    0.2164    1.1891   -0.7378   -0.6676    2.7433   -0.3158    0.5120   -2.9395
    0.9913   -2.1473   -0.5200    1.6760   -0.3158    9.2949   -2.6264   -6.3527
   -0.7378   -0.7178    0.3250    1.1306    0.5120   -2.6264    2.0654    0.0489
   -0.4698    1.6760    0.9328   -2.1391   -2.9395   -6.3527    0.0489    9.2433
```

- Due to the distorted shape of the element, the contributions of the stiffness matrix obtained using 2 and 3 points are different

Results: numerical integration

- Stiffness matrix using different integration schemes



element 33

```
* --- Using 2 x 2 points
K_el =
   1.0e+04 *
    4.5714   -3.6484   -2.2857    1.3626    1.2857   -0.0989   -1.2857    0.0989
   -3.6484    4.5714    1.3626   -2.2857    0.0989   -1.2857   -0.0989    1.2857
   -2.2857    1.3626    4.5714   -3.6484   -1.2857    0.0989    1.2857   -0.0989
    1.3626   -2.2857   -3.6484    4.5714   -0.0989    1.2857    0.0989   -1.2857
    1.2857    0.0989   -1.2857   -0.0989    3.1429   -0.5055   -1.5714   -1.0659
   -0.0989   -1.2857    0.0989    1.2857   -0.5055    3.1429   -1.0659   -1.5714
   -1.2857   -0.0989    1.2857    0.0989   -1.5714   -1.0659    3.1429   -0.5055
    0.0989    1.2857   -0.0989   -1.2857   -1.0659   -1.5714   -0.5055    3.1429

* --- Using 3 x 3 points
K_el =
   1.0e+04 *
    4.5714   -3.6484   -2.2857    1.3626    1.2857   -0.0989   -1.2857    0.0989
   -3.6484    4.5714    1.3626   -2.2857    0.0989   -1.2857   -0.0989    1.2857
   -2.2857    1.3626    4.5714   -3.6484   -1.2857    0.0989    1.2857   -0.0989
    1.3626   -2.2857   -3.6484    4.5714   -0.0989    1.2857    0.0989   -1.2857
    1.2857    0.0989   -1.2857   -0.0989    3.1429   -0.5055   -1.5714   -1.0659
   -0.0989   -1.2857    0.0989    1.2857   -0.5055    3.1429   -1.0659   -1.5714
   -1.2857   -0.0989    1.2857    0.0989   -1.5714   -1.0659    3.1429   -0.5055
    0.0989    1.2857   -0.0989   -1.2857   -1.0659   -1.5714   -0.5055    3.1429
```

- The element is regular and is integrated exactly using 2×2 points (no distinction exists with the stiffness matrix obtained with 3×3 points)

Results: numerical integration

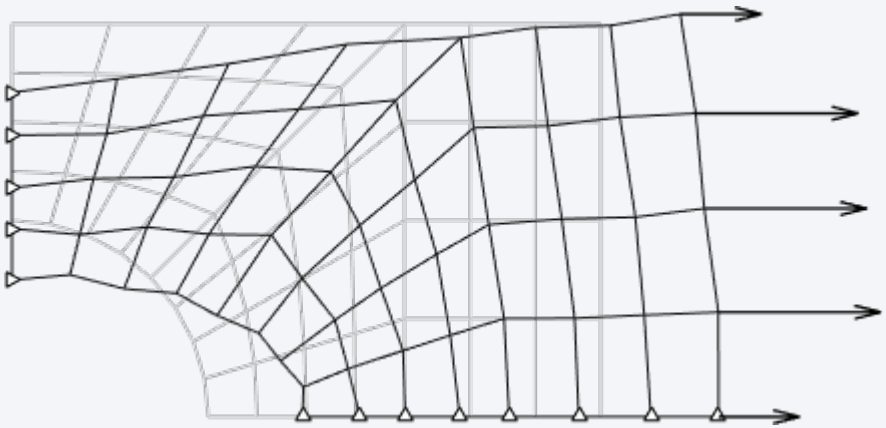
- Reported in the table are the values of the strain energy U (N mm)

	4-nodes			8-nodes		
Int rule	1×1	2×2	3×3	1×1	2×2	3×3
44 els	Hourgl.	16.1996	16.1989	/	16.5206	16.5151
570 els	16.5724	16.5499	16.5499	/	16.5776	16.5776
2400 els	16.5793	16.5738	16.5738	/	16.5807	16.5807

- For a given mesh, the effect of increasing the number of integration points is to reduce the strain energy. This means that improving the quality of the integration leads to stiffer model, viz. lower strain energy (recall that lower strain energy means a stiffer structure)
- For refined meshes (2400 els) the elements are very small, thus the distortion is slight. Improving the quality of the integration has a minor impact on the solution, as the elements are almost regular (better integration schemes are needed to capture the effects due to element distortion)

Results: numerical integration

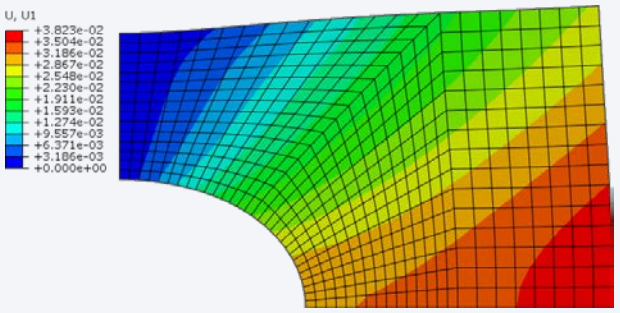
- Note that reducing the order of integration can lead to hourglassing effects, as in the case of 44 bilinear elements integrated with one single point. This is clearly visible from the deformed pattern which is characterized by the typical hourglass shape of some elements



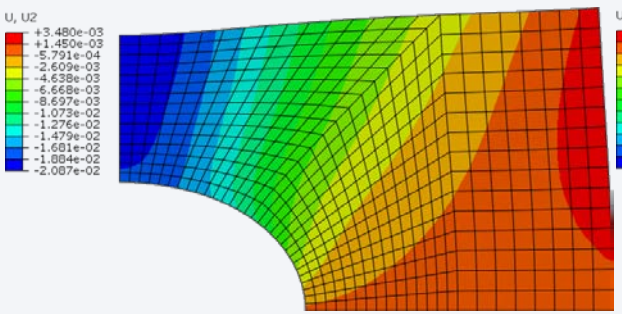
(deformed shape characterized by zero energy modes)

Results: comparison with Abaqus

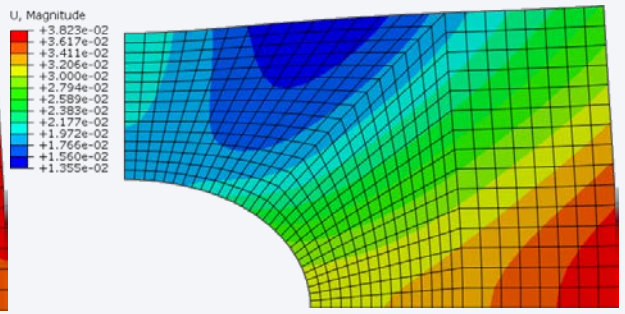
- Displacement field



U_x

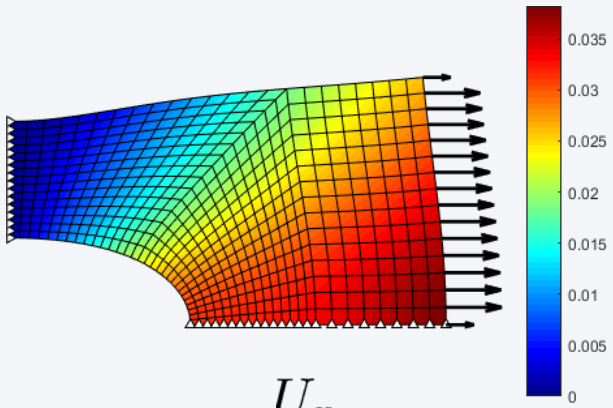


U_y

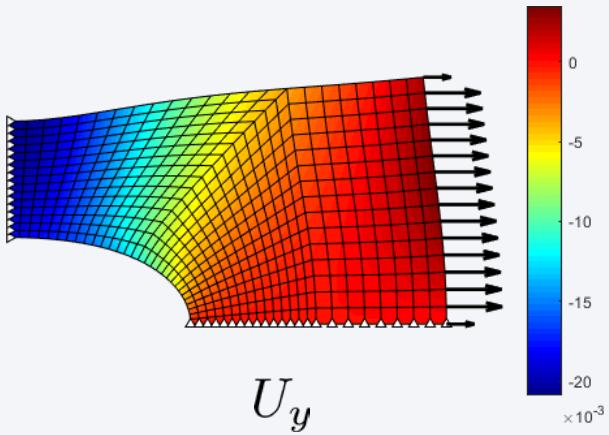


U_{tot}

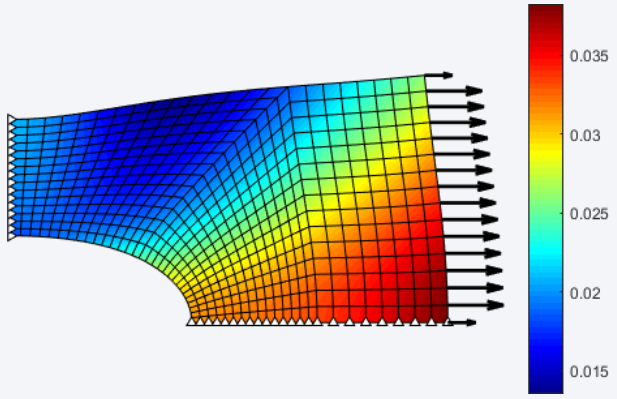
Abaqus



U_x



U_y

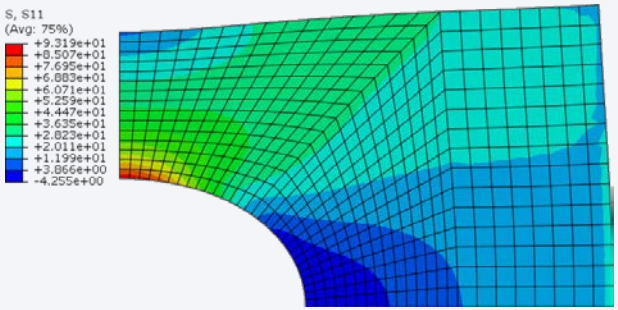


U_{tot}

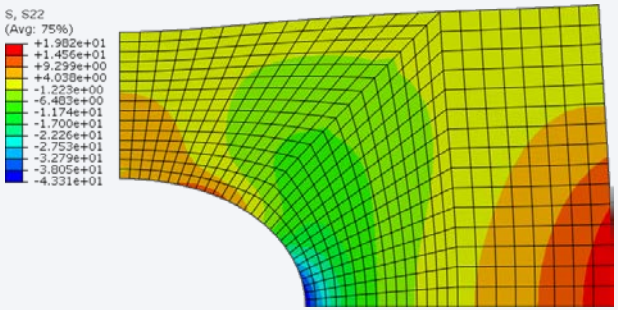
Matlab

Results: comparison with Abaqus

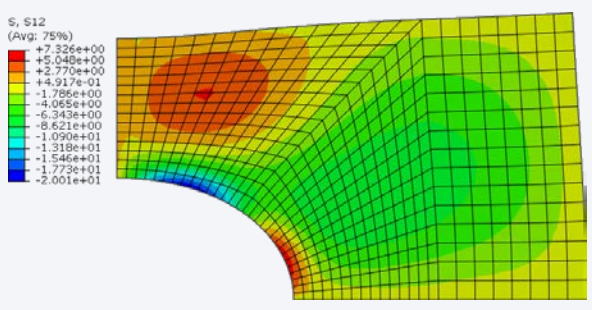
- Stress field



σ_{xx}

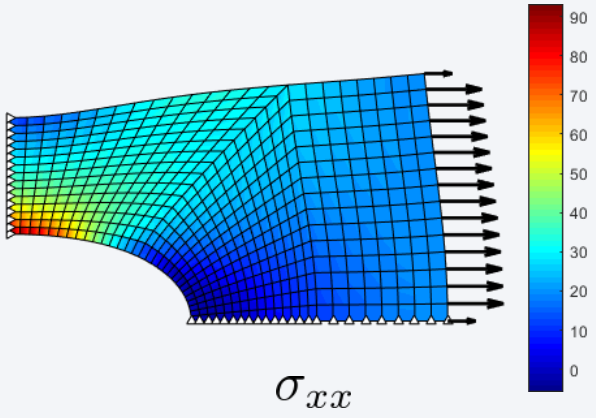


σ_{yy}

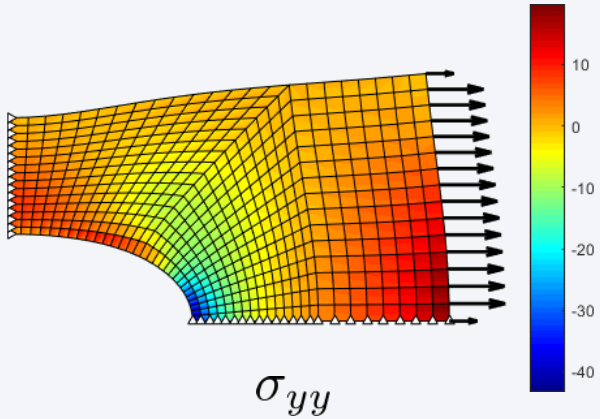


σ_{xy}

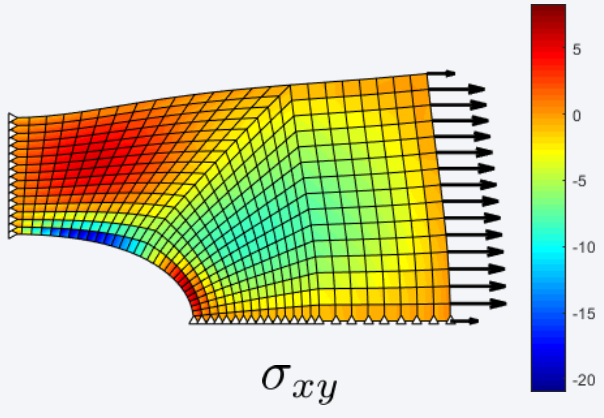
Abaqus



σ_{xx}



σ_{yy}



σ_{xy}

Matlab

Results: comparison with Abaqus

- Stresses at the nodes of element 346 (mesh 570 elements, 4-nodes)

```
THE FOLLOWING TABLE IS PRINTED AT THE NODES FOR ELEMENT TYPE M3D4 AND
ELEMENT SET EGLOBAL

    ELEMENT          ND    FOOT-    S11          S22          S12
                           NOTE
```

→ Abaqus .dat file

```
       346           1          93.19        5.183       0.6192
       346          17          92.96        5.657       -4.854
       346          18          74.91       0.1884       -4.102
       346           2          75.06      -0.2584       0.8615
```

```
>> POST.sxx(346,:)
ans =
   93.1938   75.0562   74.9058   92.9648
>> POST.syy(346,:)
ans =
    5.1829   -0.2584    0.1884    5.6567
>> POST.sxy(346,:)
ans =
    0.6192    0.8615   -4.1022   -4.8540
```

→ Matlab results