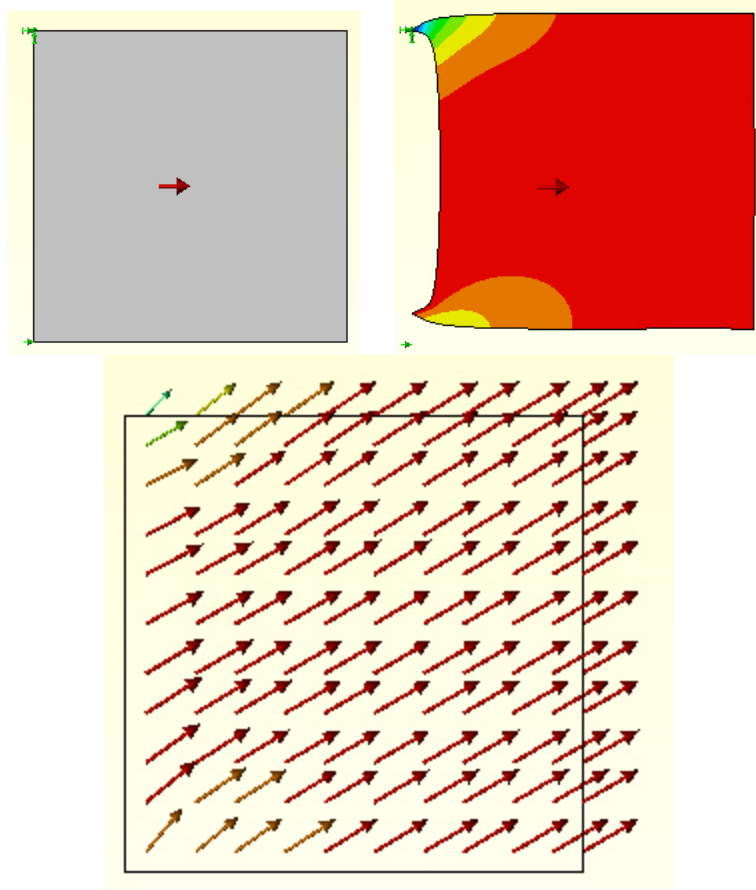


Matlab Plane Stress Example

(Draft 2, April 9, 2007)

Introduction

Here the Matlab closed form element matrices for the T3 element (3 node triangle, constant stress) is illustrated for a square plate, 2 by 2 inches. It is fixed at the top left corner, is restrained from horizontal (but not vertical) displacement at its bottom left corner. It is loaded only by a horizontal body force load acting to the right.



Away from the stress concentrations at the two corners it is essentially a 1-D problem. If the whole left edge were restrained against horizontal motion it would correspond to a axial bar hanging under its own weight. Then the free end deflection is $\delta = W L_x / (2 A E)$ and the axial stress varies from $\sigma_{\max} = W / A$ at the support to zero at the free end. Here L_x is the horizontal length, W the weight, E the elastic modulus, $A = L_x t$ is the area for a given thickness ($t=0.005$ here).

Due to Poisson's ratio, the solution here has a vertical (y) displacement. The above plot shows the plate and a finer mesh solution for the resultant displacements, from CosmosWorks. The stress in the y -direction should be zero except at the two stress concentrations.

Matlab solution

Here a crude mesh is formed by using the two diagonal lines to form four elements that are three node (constant strain) triangles, CST. There are two displacement components (degrees of freedom, dof) at each node. There are three strains and three stresses to be determined. The execution and sample plots will be shown first. The modular source script is listed at the end of this document, and as a downloadable file on the class web site. The nodal data are stored in the file `msh_bc_xyz.tmp`, the element type and connectivity in `msh_typ_nodes.tmp`, the essential (displacement component) boundary condition data in `msh_ebc.tmp`. If point loads or sources existed then file `msh_load_pt.tmp` would also be present. Those files are used for data validation plots as well as input to the stress calculations.

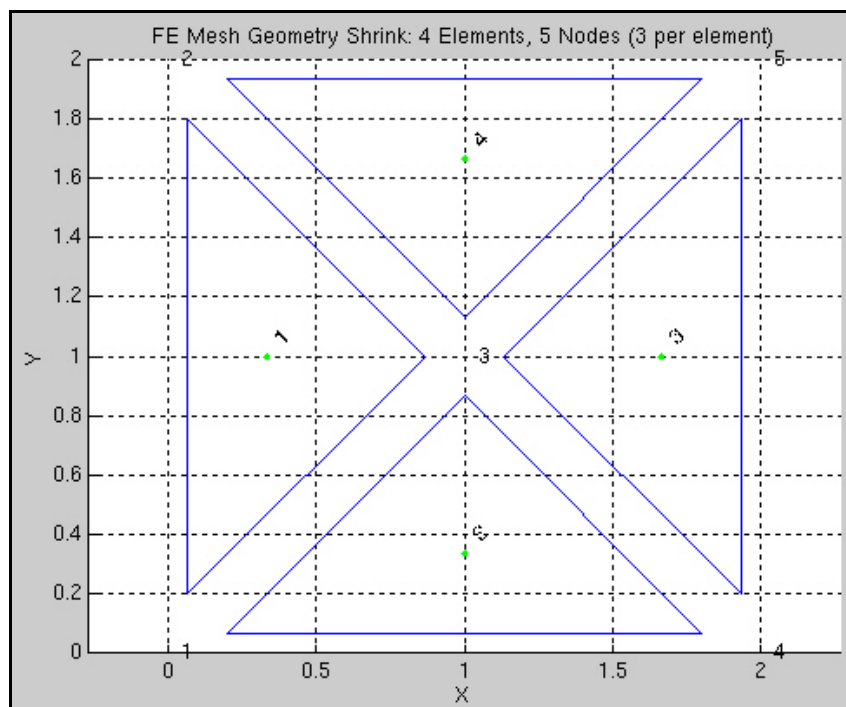
The current example starts in Unix by invoking Matlab and running mesh plotting options to check the data.

```
% Matlab
```

```
>> addpath /net/course-a/mech517/public_html/Matlab_Plots
```

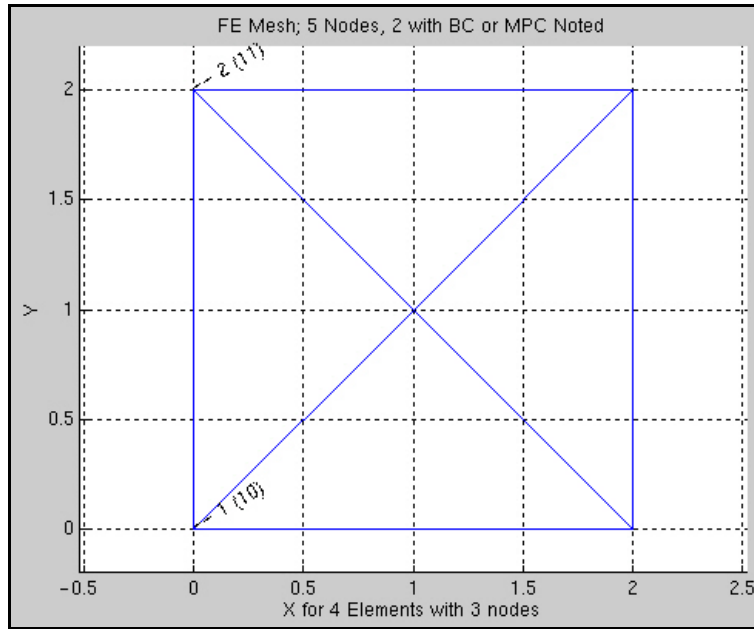
```
>> mesh_shrink_plot
```

Read 5 mesh coordinate pairs, 4 elements with 3 nodes each



```
>> bc_flags_plot
```

This displays the packed binary code for each node that has an essential boundary condition. Since there are two displacement components here there are two digits in the packed integer. A one denotes true (a restraint exists) while a zero denotes a free displacement. They are ordered as U_x (horizontal) and U_y (vertical) components. For this application the allowed flags are 00, 10, 01, or 11.



Having validated the input geometry and boundary condition flags the stress calculations are invoked:

>> Modular_Plane_Stress_XY

Read 5 nodes with bc_flag & 2 coordinates. (file **msh_bc_xyz.tmp**)

10	0	0
11	0	2
00	1	1
00	2	0
00	2	2

Read 4 elements with type number & 3 nodes each. (file **msh_typ_nodes.tmp**)

Maximum number of element types = 1.

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

Read 3 EBC data with

Node, DOF, Value. (file **msh_ebc.tmp**)

1	1	0
2	1	0
2	2	0

Application properties are:

Thickness = 0.005

Body Force = 500000 0

Elastity matrix: 1.0e+10 *

1.6000	0.4000	0
0.4000	1.6000	0
0	0	0.6000

Node	DOF	Resultant Load Value
1	1	1666.67
2	1	1666.67
3	1	3333.33
4	1	1666.67
5	1	1666.67

Displacement Solution:

X_disp	Y_disp	at 5 nodes
1.0e-04 *		
0	0.3403	
0	0	
0.5243	0.1701	
0.6667	0.1667	
0.6667	0.1736	

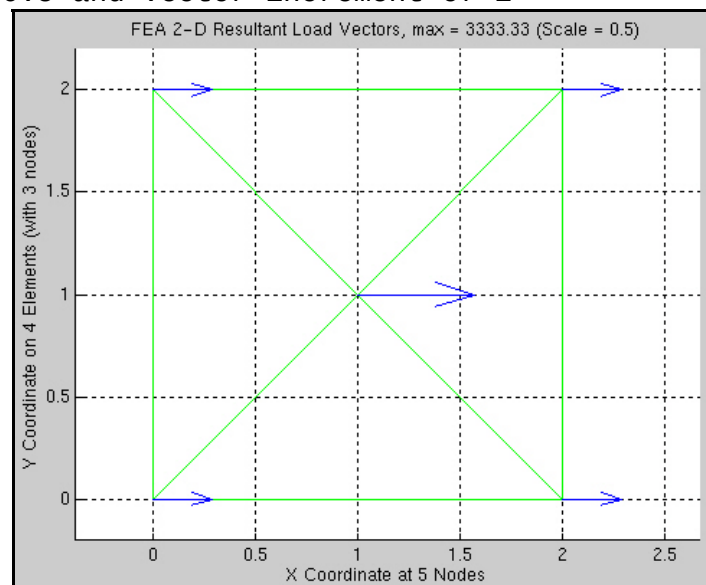
Node	DOF	Reaction Force Value
1	1	-5000
2	1	-5000
2	2	-9.37916e-13

Elem	QP	X_qp	Y_qp	(Location of stresses)
1	1	0.333333	1	
1	1	770833	-62500	1.01644e-10
2	1	1	0.333333	
2	1	500000	-5.82077e-11	62500
3	1	1.66667	1	
3	1	229167	62500	-2.03288e-11
4	1	1	1.66667	
4	1	500000	-2.91038e-11	-62500

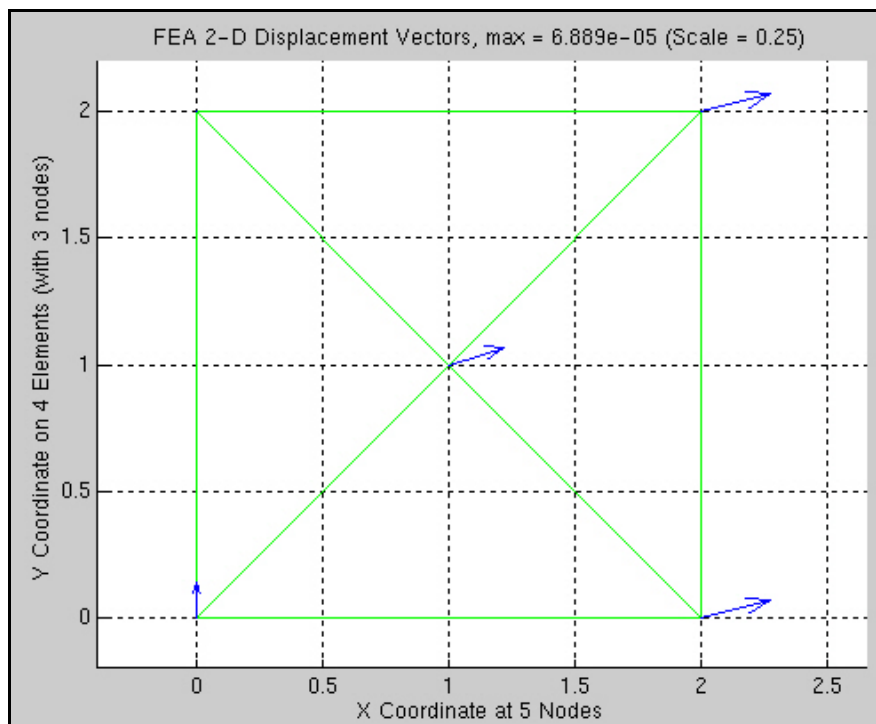
End of stress analysis. Now plot various results.

>> quiver_resultant_load_mesh(0.5)

Using a scale of 0.5 and vector increment of 1

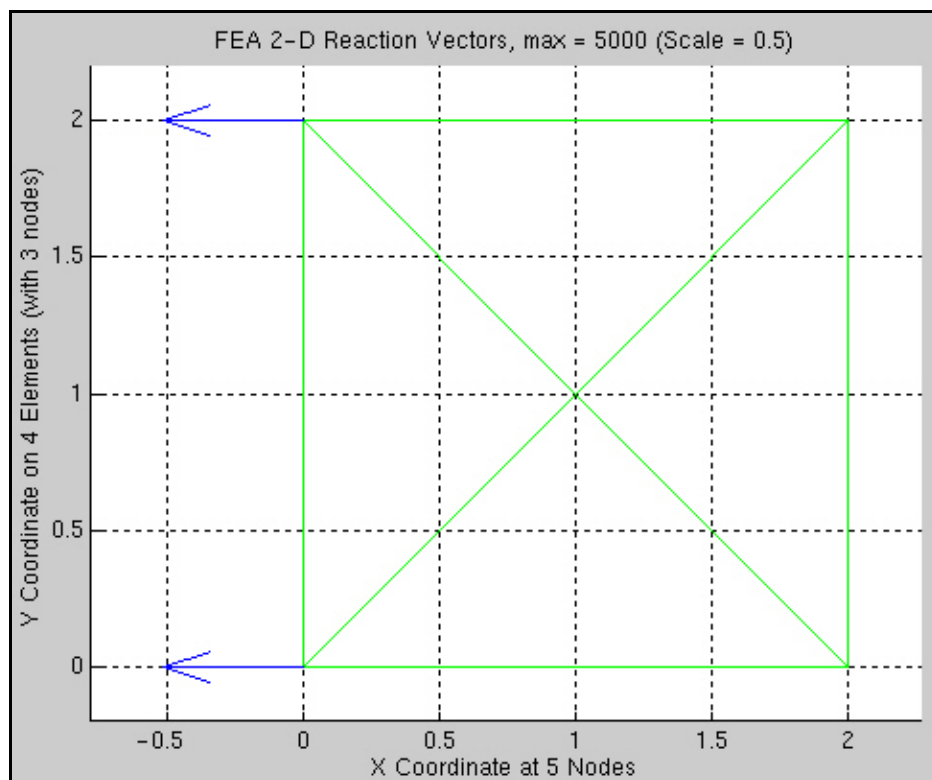


```
>> quiver_disp_vec_mesh(0.25)
```

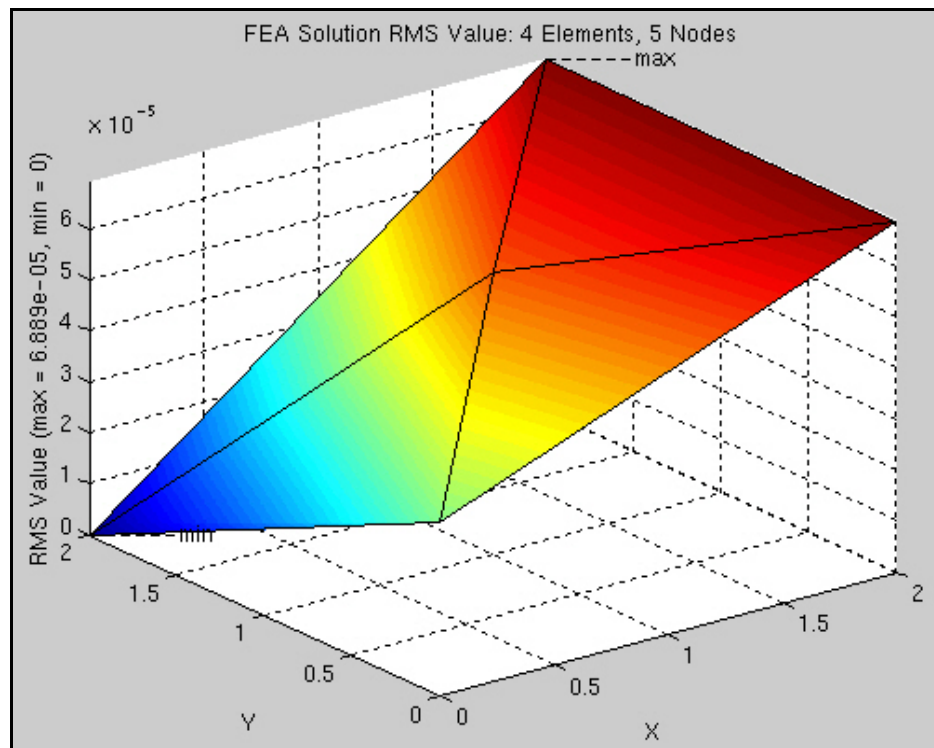


```
>> quiver_reaction_vec_mesh(0.5)
```

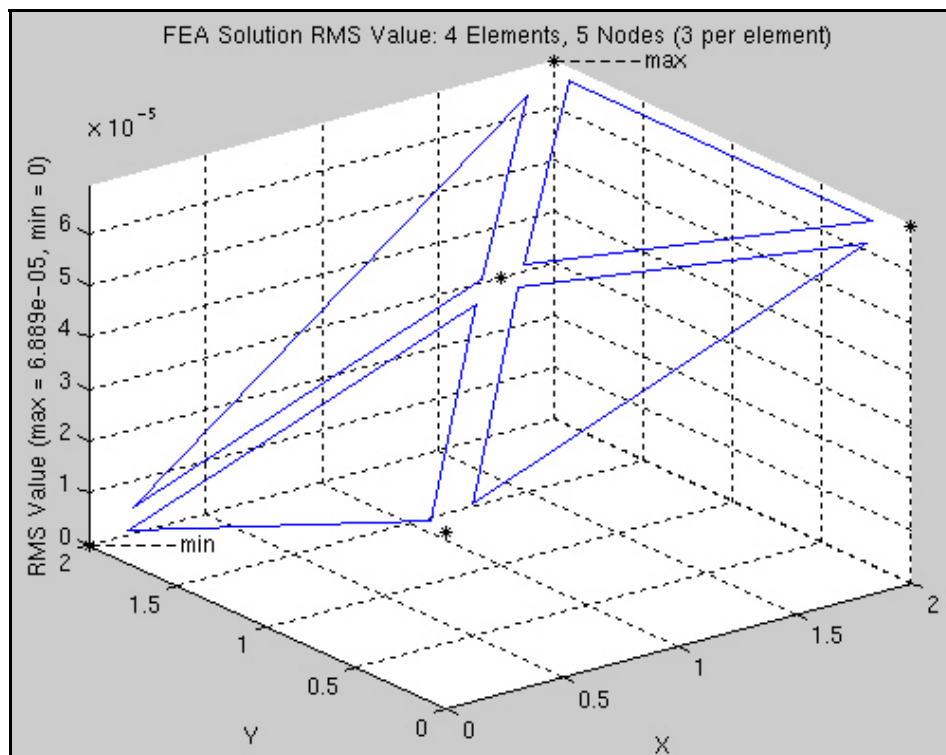
Using a scale of 0.5 and vector increment of 1



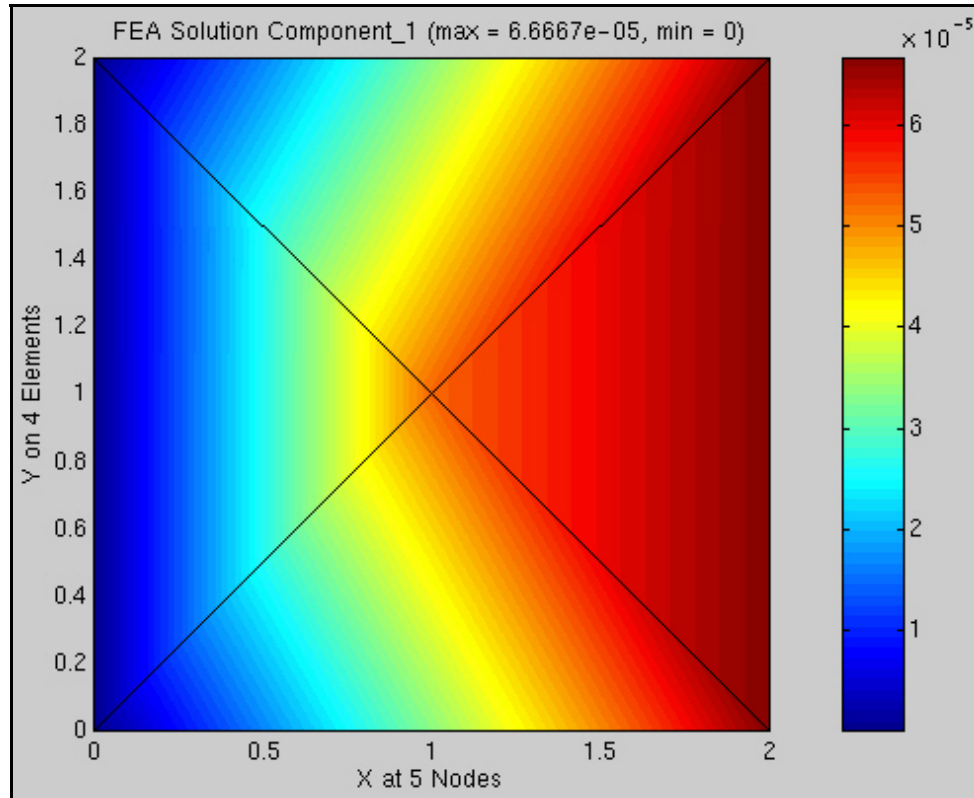
```
>> color_result_surface      (resultant displacement magnitude)
```



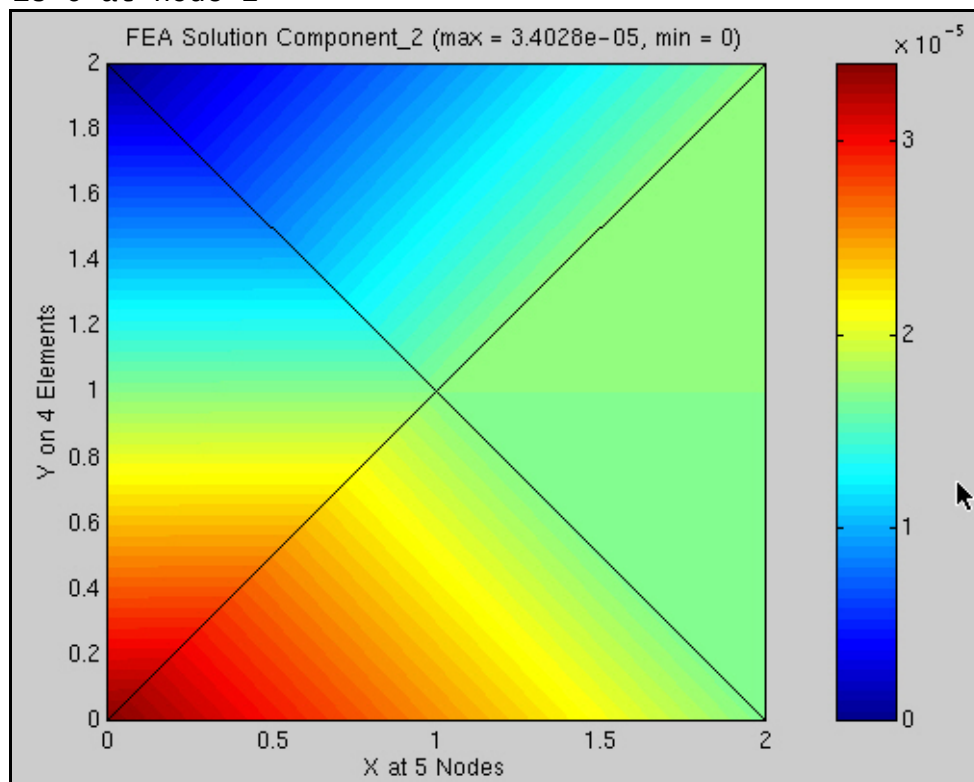
```
>> result_shrink_surface    (resultant displacement magnitude)
```



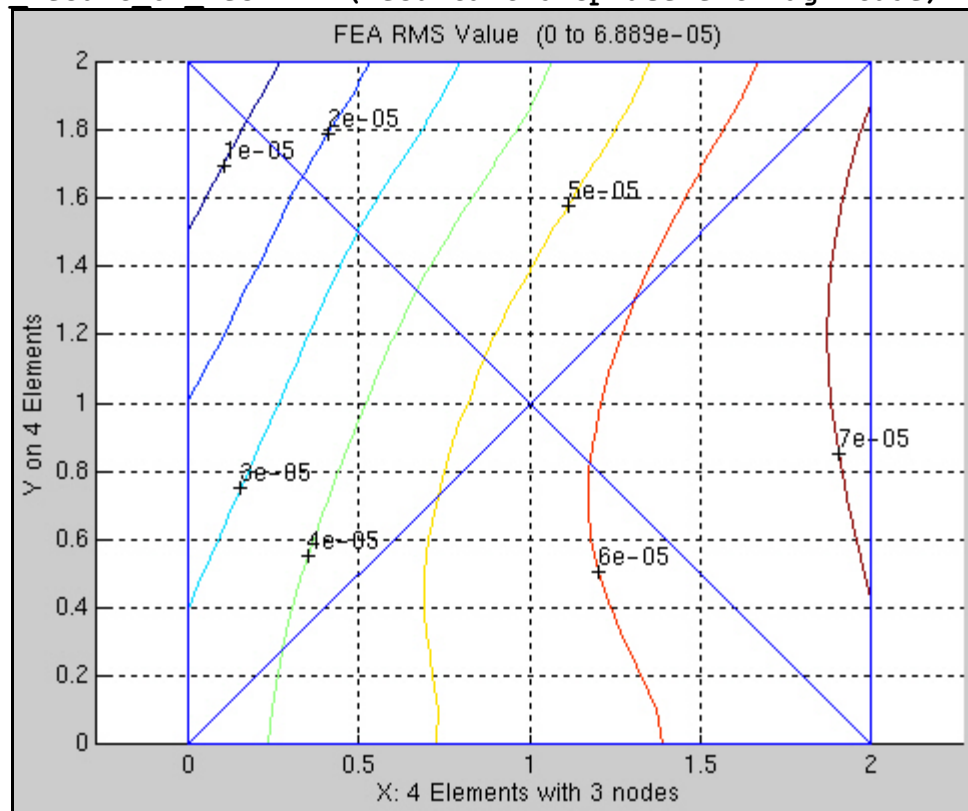
```
>> color_result(1)    (X-displacement component)
Max value is 6.66667e-05 at node 4
Min value is 0 at node 1
```



```
>> color_result(2)    ((X-displacement component)-displacement component)
Max value is 3.40278e-05 at node 1
Min value is 0 at node 2
```

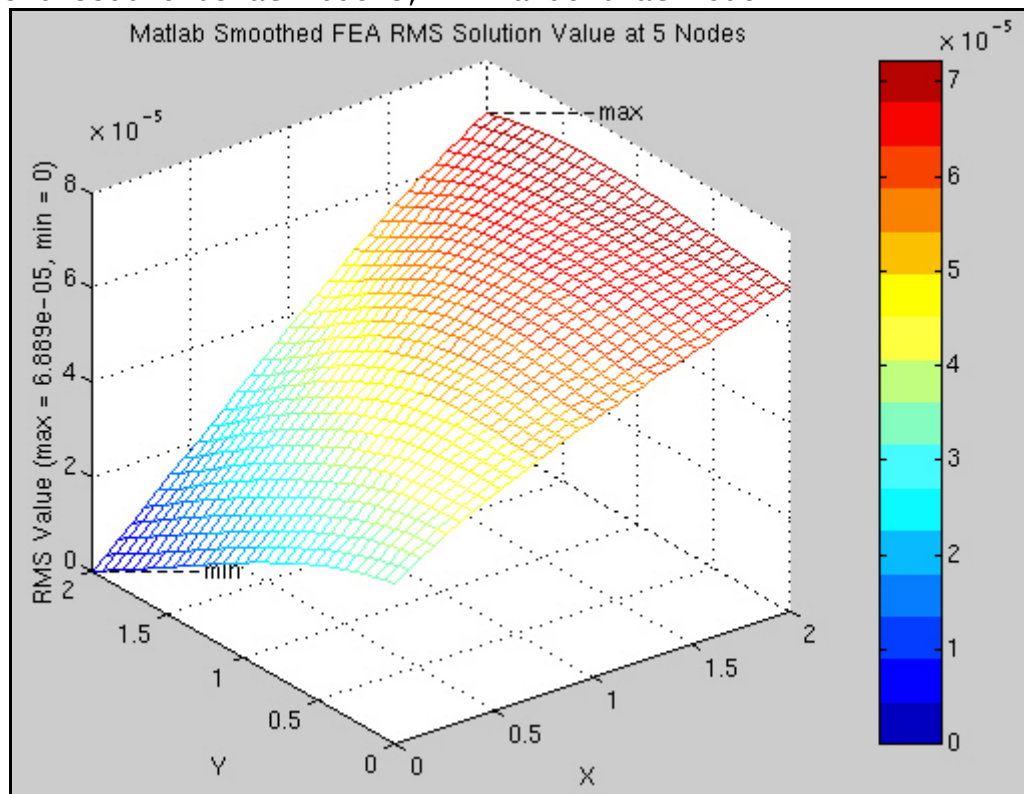


```
>> contour result on mesh (resultant displacement magnitude)
```



```
>> smooth_result_w_bar (resultant displacement magnitude)
```

Max value 6.88902e-05 at node 5, Min value 0 at node 2



```
>> quit
```

Source listing

```
function Modular_Plane_Stress_XY (load_pt, pre_p, pre_e)
%.....
%      Plane Stress with body and point loads, T3 triangle
%      XY COORDINATES CLOSED FORM INTEGRALS
%      Change E_e to get Plane Strain
%.....
% pre_e = # of dummy items before el_type & connectivity
% pre_p = # of dummy items before BC_flag % coordinates
if ( nargin == 2 )                ; % check for optional data
    pre_e = 0                    ; % no el # in msh_typ_nodes
elseif ( nargin == 1 )            ; % check for optional data
    pre_p = 0 ; pre_e = 0        ; % no pt # in msh_bc_xyz
elseif ( nargin == 0 )            ; % check for optional data
    load_pt = 0 ; pre_p = 0 ; pre_e = 0 ; % no point source data
end % if from argument count

%      Application and element dependent controls
n_g = 2                ; % number of DOF per node (temperature)
n_q = 1                ; % number of quadrature points required
n_r = 3                ; % number of rows in B_e matrix

%      Read mesh input data files
[n_m, n_s, P, x, y, z] = get_mesh_nodes (pre_p) ;
[n_e, n_n, n_t, el_type, nodes] = get_mesh_elements (pre_e) ;
n_d = n_g*n_m          ; % system degrees of freedom (DOF)
n_i = n_g*n_n          ; % number of DOF per element
S = zeros (n_d, n_d) ; C = zeros (n_d, 1)      ; % initialize sums

%      Extract EBC flags from packed integer flag P
[EBC_flag] = get_ebc_flags (n_g, n_m, P)        ; % unpack flags
EBC_count = sum( sum ( EBC_flag > 0 ) )         ; % # of EBC
fprintf ('Note: expecting %g EBC values. \n', EBC_count)

%      Read EBC values, if any
if ( EBC_count > 0 )                ; % need EBC data
    [EBC_value] = get_ebc_values (n_g, n_m, EBC_flag) ; % read data
end % if any EBC data expected

%      Read Neumann point loads data, if any, and insert in C
if ( load_pt > 0 )                ; % need point loads data
    [C] = get_and_add_point_sources (n_g, n_m, C); % add point loads
end % if any point source expected

% ===== ASSUMING HOMOGENOUS PROPERTIES =====
%      GENERATE ELEMENT MATRICES AND ASSYMBLE INTO SYSTEM
%      Assemble n_d by n_d square matrix terms from n_e by n_e

B_e = zeros (n_r, n_i)                ; % clear array
for j = 1:n_e                        ; % loop over elements =====>
    S_e = zeros (n_i, n_i)            ; % clear array
    C_b = zeros (n_i, 1) ; C_e = zeros (n_i, 1) ; % clear arrays
    e_nodes = nodes (j, 1:n_n)        ; % connectivity
```

```

%          SET ELEMENT PROPERTIES & GEOMETRY
[t_e, Body_e, E_e] = set_constant_plane_stress_prop; % properties
thick = t_e * el_type (j) ; % integer multiple
[a, b, c, center, two_A] = form_T3_geom_constants (x, y, e_nodes);

%          ELEMENT CONDUCTION AND INTERNAL SOURCE MATRICES
% for q = 1:n_q ; % Loop over element quadrature points ---->

%   H_i (x,y) = (a_i + b_i*x + c_i*y)/two_A % interpolations

%   define 3 by 6 strain-displacement matrix, B_e
B_e (1, 1:2:5) = b (1:3)/two_A ; B_e (2, 2:2:6) = c (1:3)/two_A ;
B_e (3, 1:2:5) = c (1:3)/two_A ; B_e (3, 2:2:6) = b (1:3)/two_A ;

%          stiffness matrix, with constant jacobian
S_e = ( B_e' * E_e * B_e ) * thick * two_A / 2; % stiffness

%          internal body force per unit volume, Body_e
if ( any (Body_e) ) ; % then form forcing vector
    Hi_Bx = Body_e (1) * thick * two_A / 6.0 ; % integral H_i Body x
    Hi_By = Body_e (2) * thick * two_A / 6.0 ; % integral H_i Body y
    C_b (1:2:5) = [ Hi_Bx, Hi_Bx, Hi_Bx ] ; % vector result
    C_b (2:2:6) = [ Hi_By, Hi_By, Hi_By ] ; % vector result
    C_e = C_e + C_b ; % scatter
end % if or set up properties for body force
% end % for loop over n_q element quadrature points <----

%          SCATTER TO (ASSEMBLE INTO) SYSTEM ARRAYS
% Insert completed element matrices into system matrices
[rows] = get_element_index (n_g, n_n, e_nodes); % eq numbers
S (rows, rows) = S (rows, rows) + S_e ; % add to system sq
C (rows) = C (rows) + C_e ; % add to sys column
end % for each j element in mesh <=====

%          ALLOCATE STORAGE FOR OPTIONAL REACTION RECOVERY
if ( EBC_count > 0 ) ; % reactions occur
    [EBC_row, EBC_col] = save_reaction_matrices (EBC_flag, S, C);
end % if essential BC exist (almost always true)

%          ECHO PROPERTIES
fprintf ('Application properties are: \n')
fprintf ('Thickness = %g \n', thick)
fprintf ('Body Force = '), disp (Body_e)
fprintf ('Elastity matrix:'), disp(E_e)

%          ENFORCE ESSENTIAL BOUNDARY CONDITIONS
save_resultant_load_vectors (n_g, C)
[S, C] = enforce_essential_BC (EBC_flag, EBC_value, S, C);

%          COMPUTE SOLUTION & SAVE
T = S \ C ; % Compute displacements
list_save_displacements_results (n_g, n_m, T) ; % save and print

```

```

%          OPTIONAL REACTION RECOVERY & SAVE
if ( EBC_count > 0 ) ; % reactions exist ?
    [EBC_react] = recover_reactions_print_save (n_g, n_d, ...
        EBC_flag, EBC_row, EBC_col, T); % reaction to EBC
end % if EBC exist

%          POST-PROCESS ELEMENT HEAT FLUX RECOVERY & SAVE
output_PlaneStress_stresses (n_e, n_g, n_n, n_q, nodes, x, y, T)

% End finite element calculations.
% See /home/mech517/public_html/Matlab_Plots for graphic options
% http://www.owl.net.rice.edu/~mech517/help_plot.html for help
% end of Modular_Plane_Stress_XY

% ++++++ functions in alphabetical order ++++++

function [S, C] = enforce_essential_BC (EBC_flag, EBC_value, S, C)
% modify system linear eqs for essential boundary conditions
% (by trick to avoid matrix partitions, loses reaction data)
    n_d = size (C, 1) ; % number of DOF eqs
if ( size (EBC_flag, 2) > 1 ) ; % change to vector copy
    flag_EBC = reshape ( EBC_flag', 1, n_d) ;
    value_EBC = reshape ( EBC_value', 1, n_d) ;
else
    flag_EBC = EBC_flag ;
    value_EBC = EBC_value ;
end % if
for j = 1:n_d % check all DOF for essential BC
    if ( flag_EBC (j) ) % then EBC here
% Carry known columns*EBC to RHS. Zero that column and row.
% Insert EBC identity, 1*EBC_dof = EBC_value.
        EBC = value_EBC (j) ; % recover EBC value
        C (:) = C (:) - EBC * S (:, j) ; % carry known column to RHS
        S (:, j) = 0 ; S (j, :) = 0 ; % clear, restore symmetry
        S (j, j) = 1 ; C (j) = EBC ; % insert identity into row
    end % if EBC for this DOF
end % for over all j-th DOF
% end enforce_essential_BC (EBC_flag, EBC_value, S, C)

function [a, b, c, center, two_A] = form_T3_geom_constants (x, y, e_nodes
)
% Planar 3 node triangle geometry: H_i (x,y) = (a_i + b_i*x + c_i*y)/two_
a
%
%          define nodal coordinates, ccw: i, j, k
x_e = x(e_nodes) ; y_e = y(e_nodes) ; % coord at el nodes
x_i = x_e(1) ; x_j = x_e(2) ; x_k = x_e(3) ; % change notation
y_i = y_e(1) ; y_j = y_e(2) ; y_k = y_e(3) ; % change notation

% define centroid coordinates (quadrature point)
center (1) = (x_i + x_j + x_k)/3 ;
center (2) = (y_i + y_j + y_k)/3 ;

% geometric parameters: H_i (x,y) = (a_i + b_i*x + c_i*y)/two_a
a_i = x_j * y_k - x_k * y_j ; b_i = y_j - y_k ; c_i = x_k - x_j ;
a_j = x_k * y_i - x_i * y_k ; b_j = y_k - y_i ; c_j = x_i - x_k ;
a_k = x_i * y_j - x_j * y_i ; b_k = y_i - y_j ; c_k = x_j - x_i ;

```

```

a (1:3) = [a_i, a_j, a_k] ;
b (1:3) = [b_i, b_j, b_k] ;
c (1:3) = [c_i, c_j, c_k] ;

% calculate twice element area
two_A = a_i + a_j + a_k ; % = b_j*c_k - b_k*c_j also
% end form_T3_geom_constants (x, y, e_nodes)

function [C] = get_and_add_point_sources (n_g, n_m, C)
load msh_load_pt.tmp ; % node, DOF, value (eq. number)
n_u = size(msh_load_pt, 1) ; % number of point sources
if ( n_u < 1 ) ; % missing data
error ('No load_pt data in msh_load_pt.tmp')
end % if user error
fprintf ('Read %g point sources. \n', n_u)
fprintf ('Node DOF Source_value \n')
for j = 1:n_u ; % non-zero Neumann pts
node = msh_load_pt (j, 1) ; % global node number
DOF = msh_load_pt (j, 2) ; % local DOF number
value = msh_load_pt (j, 3) ; % point source value
fprintf ('%g %g %g \n', node, DOF, value)
Eq = n_g * (node - 1) + DOF ; % row in system matrix
C (Eq) = C (Eq) + value ; % add to system column matrix
end % for each EBC
fprintf ('\n')
% end get_and_add_point_sources (n_g, n_m, C)

function [EBC_flag] = get_ebc_flags (n_g, n_m, P)
EBC_flag = zeros(n_m, n_g) ; % initialize
for k = 1:n_m ; % loop over all nodes
if ( P(k) > 0 ) ; % at least one EBC here
[flags] = unpack_pt_flags (n_g, k, P(k)) ; % unpacking
EBC_flag (k, 1:n_g) = flags (1:n_g) ; % populate array
end % if EBC at node k
end % for loop over all nodes
% end get_ebc_flags

function [EBC_value] = get_ebc_values (n_g, n_m, EBC_flag)
EBC_value = zeros(n_m, n_g) ; % initialize to zero
load msh_ebc.tmp ; % node, DOF, value (eq. number)
n_c = size(msh_ebc, 1) ; % number of constraints
fprintf ('Read %g EBC data with Node, DOF, Value. \n', n_c)
disp(msh_ebc) ; % echo input

for j = 1:n_c ; % loop over ebc inputs
node = round (msh_ebc (j, 1)) ; % node in mesh
DOF = round (msh_ebc (j, 2)) ; % DOF # at node
value = msh_ebc (j, 3) ; % EBC value
% Eq = n_g * (node - 1) + DOF ; % row in system matrix
EBC_value (node, DOF) = value ; % insert value in array
if ( EBC_flag (node, DOF) == 0 ) % check data consistency
fprintf ('WARNING: EBC but no flag at node %g & DOF %g. \n', ...
node, DOF)
%b EBC_flag (node, DOF) = 1 ; % try to recover from data error
end % if common user error
end % for each EBC

```

```

EBC_count = sum (sum ( EBC_flag > 0 )) ; % check input data
if ( EBC_count ~= n_c ) ; % probable user error
    fprintf ('WARNING: mismatch in bc_flag count & msh_ebc.tmp')
end % if user error
% end get_ebc_values

function [rows] = get_element_index (n_g, n_n, e_nodes)
rows = zeros (1, n_g*n_n) ;
for k = 1:n_n ;
    global_node = round (e_nodes (k)) ;
    for i = 1:n_g ;
        eq_global = i + n_g * (global_node - 1) ;
        eq_element = i + n_g * (k - 1) ;
        if ( eq_global > 0 )
            rows (1, eq_element) = eq_global ;
        end % if allow for omitted nodes
    end % for DOF i
end % for each element node
% end get_element_index

function [n_e, n_n, n_t, el_type, nodes] = get_mesh_elements (pre_e) ;
% MODEL input file controls (for various data generators)
if (nargin == 0) ; % default to no proceeding items in data
    pre_e = 0 ; % Dummy items before el_type & connectivity
end % if

load msh_typ_nodes.tmp ; % el_type, connectivity list (3)
n_e = size (msh_typ_nodes,1) ; % number of elements
if ( n_e == 0 ) ; % data file missing
    error ('Error missing file msh_typ_nodes.tmp')
end % if error
n_n = size (msh_typ_nodes,2) - pre_e - 1 ; % nodes per element
fprintf ('Read %g elements with type number & %g nodes each. \n', ...
        n_e, n_n)
el_type = round (msh_typ_nodes(:, pre_e+1)); % el type number >= 1
n_t = max(el_type) ; % number of element types
fprintf ('Maximum number of element types = %g. \n', n_t)
nodes (1:n_e, 1:n_n) = msh_typ_nodes (1:n_e, (pre_e+2:pre_e+1+n_n));
disp(msh_typ_nodes (:, (pre_e+1:pre_e+1+n_n))) % echo data
% end get_mesh_elements

function [n_m, n_s, P, x, y, z] = get_mesh_nodes (pre_p) ;
% MODEL input file controls (for various data generators)
if (nargin == 0) % override default
    pre_p = 0 ; % Dummy items before BC_flag % coordinates
end % if

% READ MESH AND EBC_FLAG INPUT DATA
% specific problem data from MODEL data files (sequential)
load msh_bc_xyz.tmp ; % bc_flag, x-, y-, z-coords
n_m = size (msh_bc_xyz,1) ; % number of nodal points in mesh
if ( n_m == 0 ) ; % data missing !
    error ('Error missing file msh_bc_xyz.tmp')
end % if error

```

```

n_s = size (msh_bc_xyz,2) - pre_p - 1 ; % number of space dimensions
fprintf ('Read %g nodes with bc_flag & %g coordinates. \n', n_m, n_s)
msh_bc_xyz (:, (pre_p+1))= round (msh_bc_xyz (:, (pre_p+1)));
P = msh_bc_xyz (1:n_m, (pre_p+1)) ; % integer Packed BC flag
x = msh_bc_xyz (1:n_m, (pre_p+2)) ; % extract x column
y (1:n_m, 1) = 0.0 ; z (1:n_m, 1) = 0.0 ; % default to zero

if (n_s > 1 ) ; % check 2D or 3D
    y = msh_bc_xyz (1:n_m, (pre_p+3)) ; % extract y column
end % if 2D or 3D
if ( n_s == 3 ) ; % check 3D
    z = msh_bc_xyz (1:n_m, (pre_p+4)) ; % extract z column
end % if 3D
disp(msh_bc_xyz (:, (pre_p+1):(pre_p+1+n_s))) ; % echo data
% end get_mesh_nodes

function list_save_displacements_results (n_g, n_m, T)
    fprintf ('\n') ;
    fprintf('X_disp      Y_disp      Z_disp      at %g nodes \n', n_m)
    T_matrix = reshape (T, n_g, n_m)' ; % pretty shape
    disp (T_matrix) ; % print displacements
    % save results (displacements) to MODEL file: node_results.tmp
    fid = fopen('node_results.tmp', 'w') ; % open for writing
    for j = 1:n_m ; % save displacements
        if ( n_g == 1 )
            fprintf (fid, '%g \n', T_matrix (j, 1:n_g)) ;
        elseif ( n_g == 2 )
            fprintf (fid, '%g %g \n', T_matrix (j, 1:n_g)) ;
        elseif ( n_g == 3 )
            fprintf (fid, '%g %g %g \n', T_matrix (j, 1:n_g)) ;
        elseif ( n_g == 4 )
            fprintf (fid, '%g %g %g %g \n', T_matrix (j, 1:n_g)) ;
        elseif ( n_g == 5 )
            fprintf (fid, '%g %g %g %g %g \n', T_matrix (j, 1:n_g)) ;
        elseif ( n_g == 6 )
            fprintf (fid, '%g %g %g %g %g %g \n', T_matrix (j, 1:n_g)) ;
        else
            error ('reformat list_save_displacements_results for n_g > 6.')
        end % if
    end % for j DOF
% end list_save_displacements_results (T)

function list_save_temperature_results (T)
    n_m = size (T, 1) ; % get size
    fprintf('Temperature at %g nodes \n', n_m) ; % header

    % save results (temperature) to MODEL file: node_results.tmp
    fid = fopen('node_results.tmp', 'w') ; % open for writing
    for j = 1:n_m ; % save temperature
        fprintf ( fid, '%g \n', T (j)) ; % print
        fprintf (' %g %g \n', j, T (j)) ; % sequential save
    end % for j DOF
% end list_save_temperature_results (T)

```

```

function output_PlaneStress_stresses(n_e, n_g, n_n, n_q, nodes, x,y,T)
%      POST-PROCESS ELEMENT STRESS RECOVERY & SAVE
fid = fopen('el_qp_xyz_fluxes.tmp', 'w')      ; % open for writing
fprintf ('\n')                                ; % blank line
fprintf('Elem, QP, X_qp, Y_qp \n')              ;% header
fprintf('Elem, QP, Stress_qp: xx      yy      xy \n');% header

for j = 1:n_e                                ; % loop over elements  =====>
    e_nodes = nodes (j, 1:n_n)                ; % connectivity
    [a, b, c, center, two_A] = form_T3_geom_constants (x, y, e_nodes);
    [t_e, Body_e, E_e] = set_constant_plane_stress_prop; % properties

% get DOF numbers for this element, gather solution
[rows] = get_element_index (n_g, n_n, e_nodes) ; % eq numbers
T_e = T (rows)                                ; % gather element DOF

    for q = 1:n_q                            ; % Loop over element quadrature points ---->

%      H_i (x,y) = (a_i + b_i*x + c_i*y)/two_A    % interpolations
        B_e (1, 1:2:5) = b (1:3)/two_A ; B_e (2, 2:2:6) = c (1:3)/two_A ;
        B_e (3, 1:2:5) = c (1:3)/two_A ; B_e (3, 2:2:6) = b (1:3)/two_A ;

% COMPUTE GRADIENT & HEAT FLUX, SAVE LOCATION AND VALUES
        Strains = B_e * T_e                    ; % mechanical strain
        Stress = E_e * Strains                  ; % mechanical stress
        fprintf (fid,'%g %g %g %g %g \n', center(1), center(2), ...
            Stress(1), Stress(2), Stress(3));% save
        fprintf ('%g %g %g %g \n', j, q, center(1:2));% prt
        fprintf ('%g %g %g %g %g \n', j, q, Stress(1:3));% prt
        fprintf ('\n')                          ;% prt
    end % for loop over n_q element quadrature points <----
end % for each j element in mesh
% end output_PlaneStress_stresses (n_e, n_g, n_n, n_q, nodes, x, y, T)

```

```

function output_T3_heat_flux (n_e, n_g, n_n, n_q, nodes, x, y, T)
%      POST-PROCESS ELEMENT HEAT FLUX RECOVERY & SAVE
fid = fopen('el_qp_xyz_fluxes.tmp', 'w')      ; % open for writing
fprintf ('\n')                                ; % blank line
fprintf('Elem, X_qp, Y_qp, HeatFlux_x, HeatFlux_y \n');% header

for j = 1:n_e                                ; % loop over elements  =====>
    e_nodes = nodes (j, 1:n_n)                ; % connectivity
    [a, b, c, center, two_A] = form_T3_geom_constants (x, y, e_nodes);
    [t_e, Body_e, E_e] = set_constant_plane_stress_prop; % properties

% get DOF numbers for this element, gather solution
[rows] = get_element_index (n_g, n_n, e_nodes) ; % eq numbers
T_e = T (rows)                                ; % gather element DOF

    for q = 1:n_q                            ; % Loop over element quadrature points ---->

%      H_i (x,y) = (a_i + b_i*x + c_i*y)/two_A    % interpolations
        B_e (1, 1:3) = b(1:3) / two_A          ; % dH/dx
        B_e (2, 1:3) = c(1:3) / two_A          ; % dH/dy

```

```

% COMPUTE GRADIENT & HEAT FLUX, SAVE LOCATION AND VALUES
Gradient = B_e * T_e ; % gradient vector
HeatFlux = E_e * Gradient ; % heat flux vector
fprintf (fid, '%g %g %g %g \n', center(1:2), HeatFlux(1:2));% save
fprintf ('%g %g %g %g %g \n', j, center(1:2), HeatFlux(1:2));% prt
end % for loop over n_q element quadrature points <----
end % for each j element in mesh <====
% end output_T3_heat_flux (n_e, n_g, n_n, n_q, nodes, x, y, T)

```

```

function [EBC_react] = recover_reactions_print_save (n_g, n_d, ...
            EBC_flag, EBC_row, EBC_col, T)
% get EBC reaction values by using rows of S & C (before EBC)
n_d = size (T, 1) ; % number of system DOF
% n_c x 1 = n_c x n_d * n_d x 1 + n_c x 1
EBC_react = EBC_row * T - EBC_col ; % matrix reactions (+-)
% save reactions (forces) to MODEL file: node_reaction.tmp
fprintf ('\n') ; % Skip a line
fprintf ('Node, DOF, Value, Equation Number for %g Reactions \n', ...
        sum (sum (EBC_flag > 0))) ; % header
fid = fopen('node_reaction.tmp', 'w') ; % open for writing
if ( size (EBC_flag, 2) > 1 ) ; % change to vector copy
    flag_EBC = reshape ( EBC_flag', 1, n_d) ; % changed
else
    flag_EBC = EBC_flag ; % original vector
end % if
kount = 0 ; % initialize counter
for j = 1:n_d ; % extract all EBC reactions
    if ( flag_EBC(j) ) ; % then EBC here
        % Output node_number, component_number, value, equation_number
        kount = kount + 1 ; % copy counter
        node = ceil(j/n_g) ; % node at DOF j
        j_g = j - (node - 1)*n_g ; % 1 <= j_g <= n_g
        React = EBC_react (kount, 1) ; % reaction value
        fprintf ( fid, '%g %g %g %g \n', node, j_g, React, j);% save
        fprintf ('%g %g %g %g \n', node, j_g, React, j); % print
    end % if EBC for this DOF
end % for over all j-th DOF
% end recover_reactions_print_save (EBC_row, EBC_col, T)

```

```

function [EBC_row, EBC_col] = save_reaction_matrices (EBC_flag, S, C)
n_d = size (C, 1) ; % number of system DOF
EBC_count = sum (sum (EBC_flag)) ; % count EBC & reactions
EBC_row = zeros(EBC_count, n_d) ; % reaction data
EBC_col = zeros(EBC_count, 1) ; % reaction data
if ( size (EBC_flag, 2) > 1 ) ; % change to vector copy
    flag_EBC = reshape ( EBC_flag', 1, n_d) ; % changed
else
    flag_EBC = EBC_flag ; % original vector
end % if
kount = 0 ; % initialize counter
for j = 1:n_d % check all DOF for displacement BC
    if ( flag_EBC (j) ) ; % then EBC here
        % Save reaction data to be destroyed by EBC solver trick
        kount = kount + 1 ; % copy counter
    end
end

```

```

        EBC_row(kount, 1:n_d) = S (j, 1:n_d) ; % copy reaction data
        EBC_col(kount, 1)      = C (j)          ; % copy reaction data
    end % if EBC for this DOF
end % for over all j-th DOF
% end save_reaction_matrices (S, C, EBC_flag)

```

```

function save_resultant_load_vectors (n_g, C)
    n_d = size (C, 1) ; % number of system DOF
    % save resultant forces to MODEL file: node_resultants.tmp
    fprintf ('\n') ; % Skip a line
    fprintf ('Node, DOF, Resultant Value, Equation Number \n')
    fid = fopen('node_resultant.tmp', 'w') ; % open for writing
    for j = 1:n_d ; % extract all resultants
        if ( C (j) ~= 0. ) ; % then source here
            % Output node_number, component_number, value, equation_number
            node = ceil(j/n_g) ; % node at DOF j
            j_g = j - (node - 1)*n_g ; % 1 <= j_g <= n_g
            value = C (j) ; % resultant value
            fprintf ( fid, '%g %g %g %g \n', node, j_g, value, j); % save
            fprintf ('%g %g %g %g \n', node, j_g, value, j); % print
        end % if non-zero for this DOF
    end % for over all j-th DOF
% end save_resultant_load_vectors (n_g, n_m, C)

```

```

function [t_e, Body_e, E_e] = set_constant_plane_stress_prop ;
    t_e = 1 ; Body_e (1:2) = 0. ; % defaults
%
% case 1
    t_e = 5e-3 ; % thickness
    Body_e (1:2) = [5e5, 0.] ; % components
    E = 15e9 ; % Elastic modulus
    nu = 0.25 ; % Poisson's ratio
%
% plane stress
    E_v = E/(1 - nu^2) ; % constant
    E_e (1, 1) = E_v ; E_e (1, 2) = E_v * nu ; % non-zero term
    E_e (2, 1) = E_v * nu ; E_e (2, 2) = E_v ; % non-zero term
    E_e (3, 3) = E_v * (1 - nu) / 2 ; % non-zero term
%end set_constant_plane_stress_prop

```

```

function [t_e, Q_e, E_e] = set_constant_2D_conduction_prop
    % Manually set constant element properties (Fig 11.9 text)
    Q_e = 0. ; t_e = 1. ; % defaults
% case 1
    Kx = 8. ; Ky = 8. ; Kxy = 0. ; % thermal conductivity
% case 2
    kx = 1. ; Ky = 1. ; Kxy = 0. ;
% insert
    E_e = zeros (2, 2) ; % constitutive matrix
    E_e (1, 1) = Kx ; E_e (1, 2) = Kxy ; % non-zero term
    E_e (2, 1) = Kxy ; E_e (2, 2) = Ky ; % non-zero term
% end set_constant_2D_conduction_prop

```

```

function [flags] = unpack_pt_flags (n_g, N, flag)
% unpack n_g integer flags from the n_g digit flag at node N
% integer flag contains (left to right) f_1 f_2 ... f_n_g
% integer flag contains (left to right) f_1 f_2 ... f_n_g
full = flag ; % copy integer
check = 0 ; % validate input
for Left2Right = 1:n_g ; % loop over local DOF at k
    Right2Left = n_g + 1 - Left2Right ; % reverse direction
    work = floor (full / 10) ; % work item
    keep = full - work * 10 ; % work item
    flags (Right2Left) = keep ; % insert into array
    full = work ; % work item
    check = check + keep * 10^(Left2Right - 1) ; % validate
end % for each local DOF
if ( flag > check ) ; % check for likely error
    fprintf ('WARNING: bc flag likely reversed at node %g. \n', N)
end % if likely user error
% end unpack_pt_flags

```