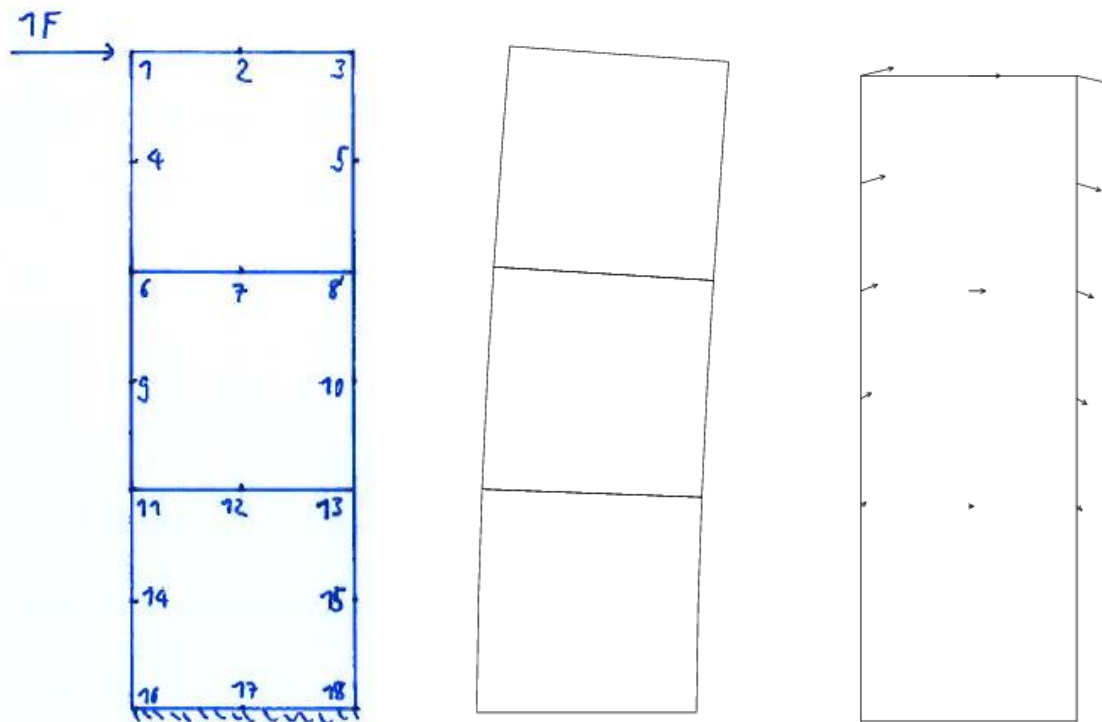


Coursework 3: 2-Dimensional Finite Element Method Programming

Author:

Tobias Schraner

student-nr.: 4990234



Abstract

This coursework's aim is to implement a simple 2-dimensional finite element program to get a better understanding of both the theory behind it and the inner workings of the FEM-programs. In a first part, the calculation is performed for one single element, which is discretised and described using linear shape functions. In another step this is expanded to a multi element problem. In a second part, quadratic shape functions are introduced to describe the elements, which imposes a greater accuracy.

Table of Contents

ABSTRACT	I
TABLE OF CONTENTS.....	II
2 SET UP OF FEM PROGRAM	3
2.1 PROGRAM STRUCTURE	3
2.2 STIFFNESS MATRIX AND ELEMENT INTEGRATION	4
2.2.1 <i>Storage</i>	5
2.3 CALCULATION OF DISPLACEMENTS.....	6
2.4 STRESSES AT INTEGRATION POINTS	6
2.5 QUADRATIC ELEMENTS.....	6
3 OUTPUT.....	7
3.1 INITIAL PROBLEM	7
3.2 MULTI-ELEMENT PROBLEM.....	7
3.3 QUADRATIC ELEMENTS	8
4 CONCLUSION	9
5 REFERENCES	10
APPENDICES.....	11
A OVERVIEW SELF-CREATED SUBROUTINES	11
B LINEAR AND QUADRATIC SHAPE FUNCTIONS AND CORRESPONDING DERIVATIVES.....	11
C SELF-CREATED SUBROUTINES	12
D MAIN CODE.....	15
E INPUT FILES	19
F OUTPUT FILES	20

1 Set up of FEM program

For the completion of this Coursework, the main programming work was performed with the program Code::Blocks, which served as an editor, debugger and compiler at the same time. The code is written in the programming language Fortran 95. Partial parts of the FEM program were provided while other parts were missing and needed to be programmed. Additionally, an example-dat-file and res-file were provided.

As in every programming task the first step is to analyse the required tasks that the program needs to perform and generate a pseudocode or program structure on how this can be implemented.

Since most problems in geo-engineering can be simplified to plane strain conditions, this is the model used for this coursework

1.1 Program structure

In Figure 1-1 the program structure is visualized. The initially missing parts are marked in a red frame.

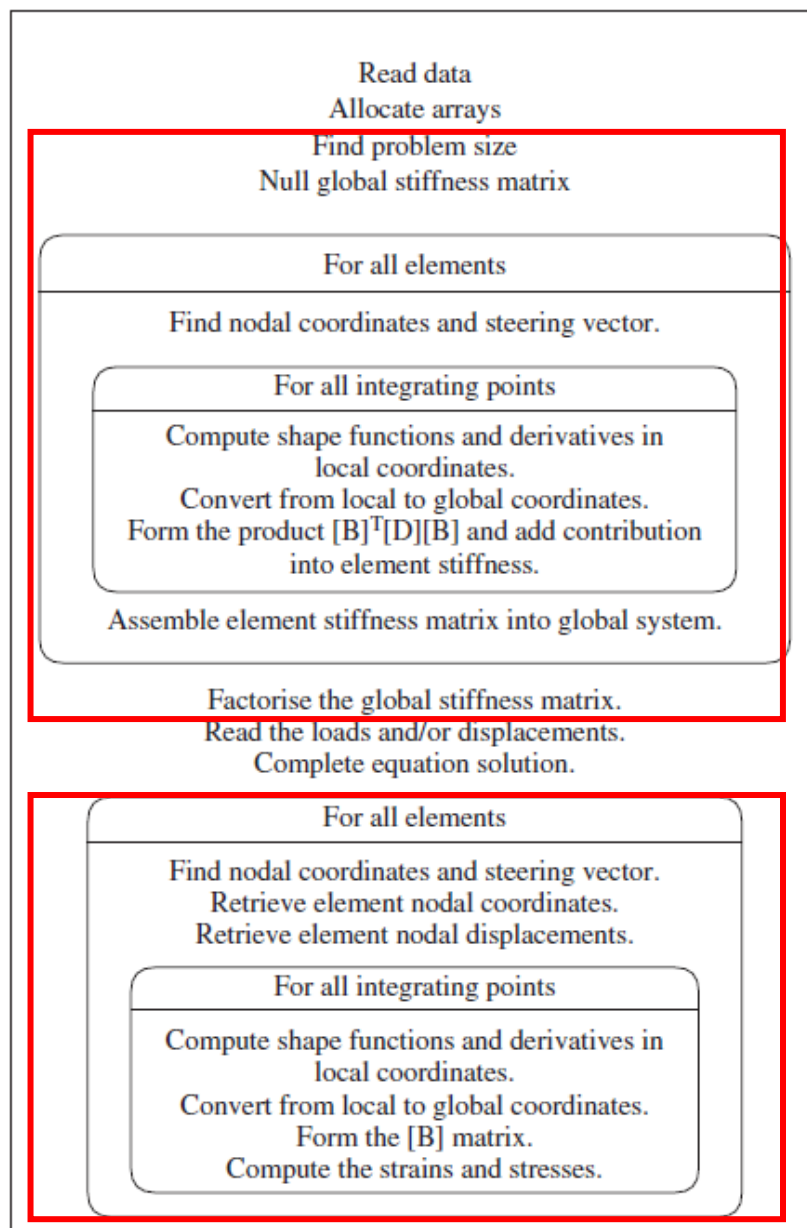


Figure 1-1 Program structure [1]

The first steps consist of defining all the variables and arrays, then reading the input data from the external input file. Subsequently, the array dimensions are searched and allocated. As a next step, local coordinates of the nodes and the gauss or integration points are created and the global stiffness matrix is nulled.

With all that prework done, every element is analysed individually in order to find the stiffness matrix. In other words, this means that with a loop it is gone through all the elements one after another. In this loop, the numbering of the nodes is transformed into local numbering. The D-matrix is created for the corresponding material properties of the element. In a inner loop which is going through all the Gauss-integration points, the numerical integration over the element is performed and the element stiffness matrix can be calculated. The stiffness matrices of all the elements are then assembled into a global system. The creation of the stiffness matrix is described in more detail in section 1.2.

After that, the loads to the system are read from the input file. With this, all the information for the solution of the governing equation are collected, the equations can be solved and the displacements can be found. Governing equation:

$$[k_m]\{u\} = \{f\}$$

These displacements are plotted in a next step for each node.

At this point, only the stresses at the integration points are left to be determined. The underlying equation is:

$$[\sigma]_i = [D]_i[B]_i\{u\}$$

This is done again with a loop for each element. In this loop the D-matrix is created for each element. Then in a loop over each integration point, the stresses are determined by calculating the equation mentioned above. The resulting stresses are then plotted. This procedure is further described in section 1.4.

1.2 Stiffness Matrix and Element Integration

For the construction of the stiffness matrix, numerical integration is needed. For this coursework, this is achieved using the Gauss Legendre quadrature. A prerequisite for this method is that the integrated functions are polynomial, which is given since the used shape functions are polynomial. The governing equation for the stiffness matrix is:

$$[k_m] = \int_0^b \int_0^a [B]^T [D] [B] dx dy = \sum_{i=1}^{nip} w_i [B]_i^T [D]_i [B]_i \det|J|_i$$

The Gauss-integration is performed on the interval $[-1,1]$. For this reason, the global coordinates of the element need to be transformed into local coordinates within the mentioned interval. The transformation is done using shape functions. Shape functions normally interpolate the solution between the discrete values obtained at the mesh nodes after the FEM-calculation but can also be used for this transformation from the local coordinates system to the global. The shape functions and their corresponding derivatives are listed in the appendix.

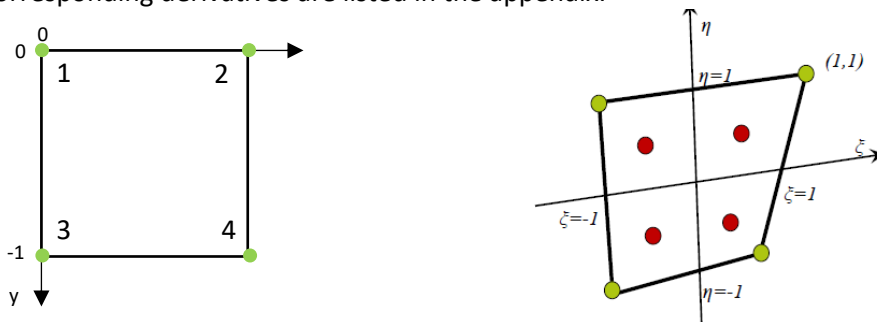


Figure 1-2 Global (left) and local numbering and coordinates (right) of the one element problem with the Gauss points indicated in red.

For the integration, the Gauss-points are needed, which are indicated red in Figure 1-2. For this first part of the coursework, 4 integration points are used while in the second part, 9 integration points are used. For the required transformation, the Jacobian matrix is needed.

$$[B]_i = [J]^{-1}[B]_{local}$$

With the Jacobian being:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

The D-matrix holds information about the material property of the element:

$$[D] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix}$$

An excerpt of the code shown in Figure 1-3 with comments on the performed action in each step.

For the generation of the local coordinates of the nodes and integration points as well as the calculation of its derivatives, two subroutines (shape_der and shape_fun) were created. Also the creation of the B-Matrix is implemented in a subroutine called beemat. More details about these subroutines can be found in the appendix.

```
call sample(coord_l, points, weights, nod) !create local xi and eta
kv=0.
Do iel=1,nels !loop for elements
  coord(:, :)=transpose(g_coord(:, g_num(:, iel))) !match input coordinates with local numbering
  call deemat(nst, prop, dee) !Create D matrix
  !print*, dee
  km = 0. !reset km for next element
  deriv = 0.
  do i=1,nip !loop for all integration points of the element
    call shape_der(coord_l, der, i, nod) !calculate local derivatives of shape functions with xi, eta
    jac=matmul(der, coord)
    !print*, "jacobian", jac
    det=determinant(jac)
    call invert(jac)
    !print*, "invert jacobian", jac
    deriv=matmul(jac, der) !calculate global derivatives of shape functions with x and y
    !print*, "deriv", deriv
    call beemat(deriv, bee, nod) !create B matrix
    !print*, "beemat", bee
    km=km+det*matmul(transpose(bee), dee, bee)*weights(i) !assemble stiffness matrix km
    !print*, "km", km
  end do

  !print*, "g_g", g_g(:, 1)
  call fsparv(kv, km, g_g(:, iel), kdiag) !finds equations that need to be solved and stores them
  !in a lower triangular matrix kv
  !assembles it into global stiffness matrix kv

  !print*, "kv", kv
  !print*, "kdiag", kdiag
end do
```

Figure 1-3 Excerpt of code where the numerical integration for the assembly of the stiffness matrix is performed.

This numerical integration is performed for every integration point to account for the effect each integration point has on the nodes and finally, the element stiffness matrix can be assembled. This integration is completed for each element with an outer do-loop.

1.2.1 Storage

For multiple elements, the global stiffness matrix can be assembled out of the individual element stiffness matrices. This global stiffness matrix can become considerably big. For one element with each node having two degrees of freedom, already a 8x8 matrix is the result. To save both storage space and computational outlay, only the equations with the unknown variables are stored. Furthermore, the global stiffness matrix kv is stored as a vector in skyline form which reduces the required storage space substantially.

1.3 Calculation of Displacements

With the calculated stiffness matrix, the equations can be solved in order to get the searched displacements.

$$[k_m]\{u\} = \{f\}$$

The used subroutines for this calculation were provided and are called sparin and spabac with the resulting displacements stored in the array 'loads'. These displacements are written into the output file with the following code lines in Figure 1-4

```
WRITE(11, '(A)') " Node x-disp y-disp"
DO j=1,nn !resulting displacements are stored in loads -> write into res file
WRITE(11, '(I5,2E10.4)') j,loads(nf(:,j))
END DO
```

Figure 1-4 code to print the calculated displacement of the nodes.

1.4 Stresses at Integration Points

The last task of the program is to recover the stresses at the gauss points. The underlying equation is:

$$[\sigma]_i = [D]_i[B]_i\{u\}$$

The code lines used for this calculation are displayed in Figure 1-5. In comments, the action at each step is described. While the outer do loop addresses all elements, the inner do loop goes through each integration points of the current element

```
DO iel=1,nels !loop for all elements
CALL deemat(nst,prop,dee) !construct D matrix
eld=0.
eld=loads(g_g(:,iel)) !write resulting displacements in eld
coord=TRANPOSE(g_coord(:,g_num(:,iel))) !match global numbering of the element with local numbering
DO i=1,nip !loop for all integration points
CALL shape_fun(coord_l,fun,i,nod) !get N_l-N_i for transofrmation of coordinates of integration points
CALL shape_der(coord_l,der,i,nod) !calculate shape-function-derivatives with local coord of integr points
gc=MATMUL(fun,coord) !calculate global coord of integr points
jac=MATMUL(der,coord) !create jacobian
CALL invert(jac)
deriv=MATMUL(jac,der) !create global derivatives of shape functions
CALL beemat(deriv,bee,nod) !create B matrix
sigma=MATMUL(dee,MATMUL(bee,eld)) !calculate sigma
WRITE(11, '(I5,6E10.4)') iel,gc,sigma !plot resulting stresses at integration points
END DO
END DO
```

Figure 1-5 Code to calculate the stresses at the integration points.

1.5 Quadratic Elements

Quadratic elements have 8 points instead of 4. This means that each side of the element is approximated by three nodes. This additional number of nodes as well as the quadratic nature of the shape functions allows for better approximation. The difference between linear and quadratic shape functions is shown in Figure 1-6. However, the computational outlay is higher since more nodes imply more total degrees of freedom and consequently more equations to solve. The equations of the shape functions and the corresponding derivatives are listed in the appendix.

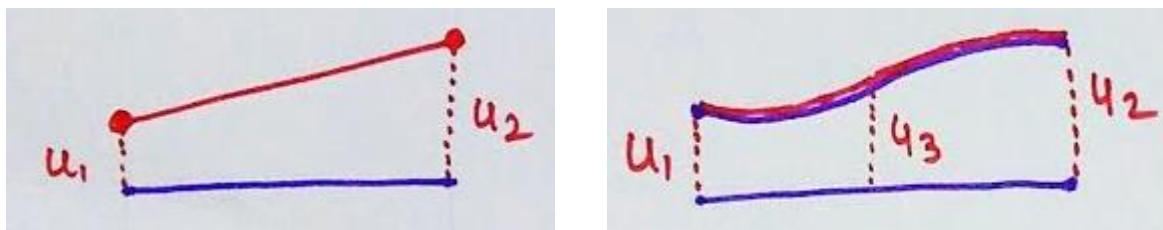


Figure 1-6 linear (left) and quadratic (right) shape functions [2]

2 Output

2.1 Initial Problem

The first problem was a one element problem with 4 nodes subjected to elastic compression. This problem is as simple as possible to verify if the code is working correctly. After the programs working is verified, it can be expanded to multiple elements. A sketch of the problem is displayed in Figure 2-1.

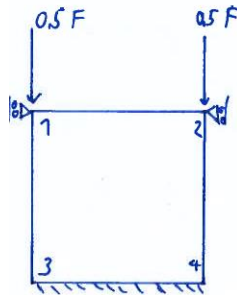


Figure 2-1 Sketch of initial problem with one element.

The nodes 1 and 2 are fixed in x-direction to make sure there is no expansion taking place in that direction. The vertical displacement of the nodes 1 and 2 count to $7.4e-7$. The full numerical results are shown in the appendix.

2.2 Multi-Element Problem

The next problem to test the programs functionality for multiple elements is a 4 element column subjected to the same load as in the initial problem. The sketch is shown in Figure 2-2. The nodes are again fixed in x-direction. As a second problem, also a bending stress was applied to the system. This will be compared to the same scenario with quadratic elements in section 2.3.

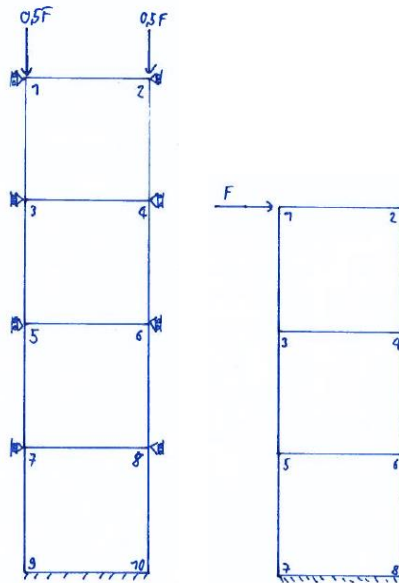


Figure 2-2 Sketches of multi element problems.

For this multi element problems, one challenge is to automatize the matching of the global numbering of the nodes to the local numbering. This has to be repeated for each element and is achieved with this intelligent line of code shown in Figure 2-3.

```
coord(:, :)=transpose(g_coord(:, g_num(:, iel)))
```

Figure 2-3 Codeline to match global numbering to local.

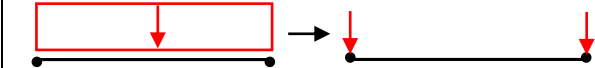
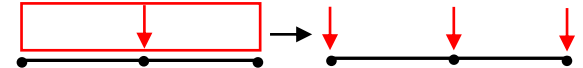
The displacements on the nodes 7 and 8 match the displacements from the first problem with $7.4e-7$. This is correct since the load is the same on this element. These displacements add up for every element which finally results in a vertical displacement of node 1 and 2 of $2.97e-6$ which is equal to $4 \cdot 7.4e-7$ and therefore is correct.

2.3 Quadratic elements

In the next part of this coursework, quadratic shape functions were introduced and applied to the one element initial problem. To make the results between the linear and the quadratic discretisation comparable, the load needs to be applied the same way. The loads in the initial problem can be seen as an equally distributed load between the two top nodes.

To discretise this distributed load into three points, the quadratic shape functions can be used. The calculation of these point loads is shown in Table 1.

Table 1 Calculation of loads for initial problem with a 8 node quadratic element.

			
$f=1$		$N_1 = \frac{\xi(\xi-1)}{2}$	
$N_1 = \frac{1-\xi}{2}$		$N_2 = (1+\xi)(1-\xi)$	
$N_2 = \frac{2+\xi}{2}$		$N_3 = \frac{\xi(\xi+1)}{2}$	
$F_1 = \int_0^l N_1 d\xi = \frac{1}{2}$		$F_1 = \int_0^l N_1 d\xi = \frac{1}{6}$	
$F_2 = \int_0^l N_2 d\xi = \frac{1}{2}$		$F_2 = \int_0^l N_2 d\xi = \frac{2}{3}$	
		$F_3 = \int_0^l N_3 d\xi = \frac{1}{6}$	

Again, the same displacement can be observed with $7.4e-7$ in points 1, 2 and 3.

Since the quadratic shape functions are able to approximate curved shapes much better, the bending stress was applied to a quadratic, 3 element column which is shown in Figure 2-4.

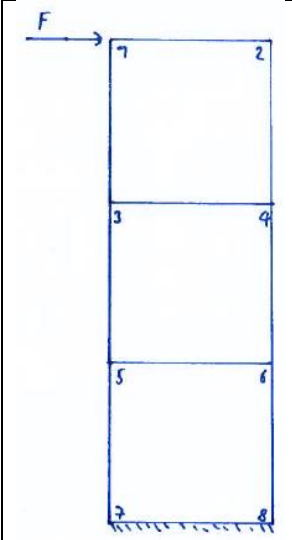
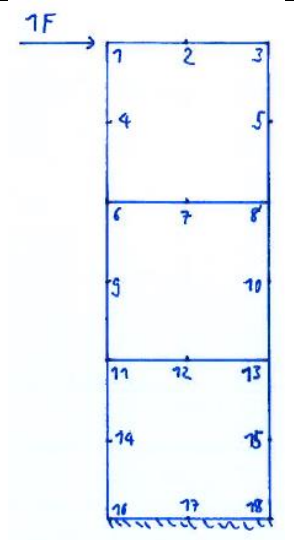
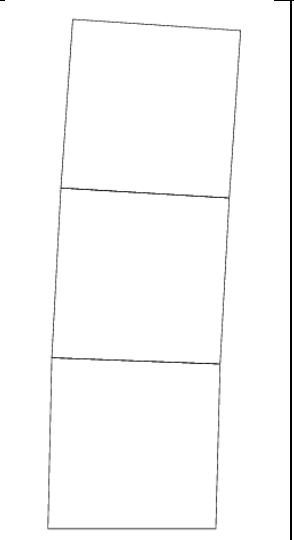
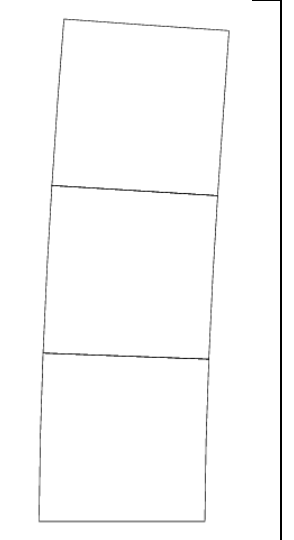
							
Linear elements		Quadratic elements		Linear elements displ.		Quadr elements displ.	

Figure 2-4 comparison between linear and quadratic elements subjected to bending

With a look at the bended column, no difference can be seen between these two element types. However, with a look at the horizontal displacements of node 1, it can be observed that the displacement of the quadratic type is larger with $1.0e-4$ compared to $0.7e-4$. Also, the nodes top nodes of the second element are more displaced in the quadratic element. To see which model is more realistic, a comparison to the analytical solution would be required but is not performed during this coursework.

With quadratic elements, 30 equations are to be solved while with linear elements only 13 equations need to be solved. On that note, it can be seen that the computational outlay and also the required storage rise fast with more nodes and it needs to be estimated which level of detail is needed for a specific problem.

3 Conclusion

In a simplified way, the chronological steps in a simple finite element calculation program are to read the input file, allocate the storage, run through each element to attain the stiffness matrix, before reading the loads or prescribed displacements and solving the equations. At this point, the resulting loads or displacements are known and can be plotted before recovering the stresses at certain points. This coursework has shown that with once the code for one element is defined, it is relatively easy scalable to larger and more detailed models and problems. However, an appropriate degree of detail should be maintained since computational costs and storage can get large very fast.

4 References

- [1] G. M. Smith, Programming the Finite Element Method, Wiley, 2014.
- [2] „Code_Aster,“ [Online]. Available: https://www.code-aster.org/V2/doc/v11/en/man_r/r3/r3.01.01.pdf. [Zugriff am 3 2019].
- [3] „wikibooks,“ [Online]. Available: https://en.wikibooks.org/wiki/Fortran/Beginning_Fortran. [Zugriff am 3 2019].
- [4] M. A. Projects. [Online]. Available: <https://www.youtube.com/watch?v=XYbyuaYVQb8>. [Zugriff am 3 2019].

Appendices

A Overview self-created Subroutines

Subroutine Name	Input	Output	description
Shape_fun (coord_l,fun,i,nod)	l,nod,coord_l	fun	Returns the shape functions N1-Nnod at the ith integrating point for the local coordinates of the integrating points(coord_l) for linear or quadratic elements.
Shape_der (coord_l,der,i,nod)	coord_l,i,nod	der	Returns the shape function derivatives at the ith integrating point for the local coordinates of the integrating points(coord_l) for linear or quadratic elements.
Sample (coord_l, points,weights,nod)	nod	coord_l, points, weights	Creates the local coordinates of the integration points (coord_l), the local coordinates of the nodes (points) and weighting for linear or quadratic elements.
Beemat (deriv,bee,nod)	nod, deriv	bee	Returns the bee matrix for shape function derivatives.
Deemat (nst, prop, dee)	nst, prop	dee	Returns elastic stress-strain dee matrix in 2D (plane strain) or 3D. E and v are Young's modulus and Poisson's ratio.

B Linear and quadratic shape functions and corresponding derivatives

N	$\partial N / \partial \xi$	$\partial N / \partial \eta$
$(1-\xi)(1-\eta)/4$	$-(1-\eta)/4$	$-(1-\xi)/4$
$(1+\xi)(1-\eta)/4$	$(1-\eta)/4$	$-(1+\xi)/4$
$(1+\xi)(1+\eta)/4$	$(1+\eta)/4$	$(1+\xi)/4$
$(1-\xi)(1+\eta)/4$	$-(1+\eta)/4$	$(1-\xi)/4$

N	$\partial N / \partial \xi$	$\partial N / \partial \eta$
$(1-\xi)(1-\eta)(-1-\xi-\eta)/4$	$(1-\eta)(2\xi+\eta)/4$	$(1-\xi)(\xi+2\eta)/4$
$(1+\xi)(1-\eta)(-1+\xi-\eta)/4$	$(1-\eta)(2\xi-\eta)/4$	$-(1+\xi)(\xi-2\eta)/4$
$(1+\xi)(1+\eta)(-1+\xi+\eta)/4$	$(1+\eta)(2\xi+\eta)/4$	$(1+\xi)(\xi+2\eta)/4$
$(1-\xi)(1+\eta)(-1-\xi+\eta)/4$	$-(1+\eta)(-2\xi+\eta)/4$	$(1-\xi)(-\xi+2\eta)/4$
$(1-\xi^2)(1-\eta)/2$	$-\xi(1-\eta)$	$-(1-\xi^2)/2$
$(1+\xi^2)(1-\eta^2)/2$	$(1-\eta^2)/2$	$-\eta(1+\xi)$
$(1-\xi^2)(1+\eta)/2$	$-\xi(1+\eta)$	$(1-\xi^2)/2$
$(1-\xi)(1-\eta^2)/2$	$-(1-\eta^2)/2$	$-\eta(1-\xi)$

C Self-created subroutines

module CW3self

! This module contains the subroutines available for completing Coursework 3

! Do not modify these subroutines.

CONTAINS

SUBROUTINE Shape_fun(coord_l,fun,i,nod)

! Returns the shape functions at the ith integrating point for the local coordinates of the integrating points

IMPLICIT NONE

integer, INTENT (IN):: i, nod

REAL*8,INTENT(IN):: coord_l(:,i)

REAL*8,INTENT(OUT):: fun(i)

if (nod==4) then

!calculate N1-N4 with local coordinates xi and eta of gausspoints

fun(1)= 1.0/4.0*(1.0-coord_l(i,1))*(1.0-coord_l(i,2))

fun(2)= 1.0/4.0*(1.0-coord_l(i,1))*(1.0+coord_l(i,2))

fun(3)= 1.0/4.0*(1.0+coord_l(i,1))*(1.0+coord_l(i,2))

fun(4)= 1.0/4.0*(1.0+coord_l(i,1))*(1.0-coord_l(i,2))

else

fun(1)=1.0/4.0*(1.0-coord_l(i,1))*(1.0-coord_l(i,2))*(-coord_l(i,1)-coord_l(i,2)-1.)

fun(2)=1.0/2.0*(1.0-coord_l(i,1))*(1.0-coord_l(i,2)**2.)

fun(3)=1.0/4.0*(1.0-coord_l(i,1))*(1.0+coord_l(i,2))*(-coord_l(i,1)+coord_l(i,2)-1.)

fun(4)=1.0/2.0*(1.0-coord_l(i,1)**2)*(1.0+coord_l(i,2))

fun(5)=1.0/4.0*(1.0+coord_l(i,1))*(1.0+coord_l(i,2))*(coord_l(i,1)+coord_l(i,2)-1.)

fun(6)=1.0/2.0*(1.0+coord_l(i,1))*(1.0-coord_l(i,2)**2.)

fun(7)=1.0/4.0*(1.0+coord_l(i,1))*(1.0-coord_l(i,2))*(coord_l(i,1)-coord_l(i,2)-1.)

fun(8)=1.0/2.0*(1.0-coord_l(i,1)**2)*(1.0-coord_l(i,2))

end if

RETURN

END SUBROUTINE Shape_fun

subroutine sample(coord_l, points,weights,nod)

Implicit none

integer, INTENT (IN):: nod

REAL*8,INTENT(out):: coord_l(:,i), points(:,i),weights(i)

integer :: j

if (nod==4) then

!creating local coordinates of gausspoints

coord_l(1,:)=/-(1./SQRT(3.)),-(1./SQRT(3.))/

coord_l(2,:)=/-(1./SQRT(3.)),(1./SQRT(3.))/

coord_l(3,:)=/(1./SQRT(3.)),(1./SQRT(3.))/

coord_l(4,:)=/(1./SQRT(3.)),-(1./SQRT(3.))/

!Creating local coordinates of nodes

points=coord_l/(1./SQRT(3.))

!defining weights

weights(:)=1.

else

!Creating local coordinates of nodes

points(1,:)=/(-1.,-1.)

points(2,:)=/(-1.,0.)

```

points(3,:) = (/ -1., 1. /)
points(4,:) = (/ 0., 1. /)
points(5,:) = (/ 1., 1. /)
points(6,:) = (/ 1., 0. /)
points(7,:) = (/ 1., -1. /)
points(8,:) = (/ 0., -1. /)
!creating local coordinates of gausspoints
do j=1,nod
  coord_l(j,:) = (points(j,:) * SQRT(3./5.))
end do
coord_l(9,:) = (/ 0., 0. /)
!defining weights
weights(1) = 25./81.
weights(2) = 40./81.
weights(3) = 25./81.
weights(4) = 40./81.
weights(5) = 25./81.
weights(6) = 40./81.
weights(7) = 25./81.
weights(8) = 40./81.
weights(9) = 64./81.
end if
return
end subroutine sample

subroutine shape_der(coord_l, der, i, nod) !coord_l(i,1), coord_l(i,2), der
IMPLICIT NONE
integer, INTENT (IN) :: i, nod
REAL*8, INTENT (IN) :: coord_l(:, :) !coord_l(i,1), coord_l(i,2) !xi(:), eta(:)
REAL*8, INTENT (OUT) :: der(:, :)
Real*8 :: xi, eta
!calculate the derivatives der(localnr, 1=xi/2=eta)
if (nod==4) then
  der(1,1) = -1.0/4.0*(1.0-coord_l(i,2))
  der(1,2) = -1./4.*(1.+coord_l(i,2))
  der(1,3) = 1./4.*(1.+coord_l(i,2))
  der(1,4) = 1./4.*(1.-coord_l(i,2))

  der(2,1) = -1./4.*(1.-coord_l(i,1))
  der(2,2) = 1./4.*(1.-coord_l(i,1))
  der(2,3) = 1./4.*(1.+coord_l(i,1))
  der(2,4) = -1./4.*(1.+coord_l(i,1))
else
  xi = coord_l(i,1)
  eta = coord_l(i,2)
  der(1,1) = 1.0/4.0*(1.0-eta)*(2.*xi+eta)
  der(1,2) = -1.0/2.0*(1.-eta**2)
  der(1,3) = -1.0/4.0*(1.+eta)*(-2*xi+eta)
  der(1,4) = -xi*(1.+eta)
  der(1,5) = 1./4.*(1.+eta)*(2*xi+eta)
  der(1,6) = 1./2.*(1.-eta**2)
  der(1,7) = 1./4.*(1.-eta)*(2.*xi-eta)
  der(1,8) = -xi*(1.-eta)

```

```

    der(2,1)=1.0/4.0*(1.0-xi)*(2.*eta+xi)
    der(2,2)=-eta*(1.-xi)
    der(2,3)=1.0/4.0*(1.-xi)*(-xi+2.*eta)
    der(2,4)=1.0/2.0*(1.-xi**2)
    der(2,5)=1./4.*(1.+xi)*(xi+2.*eta)
    der(2,6)=-eta*(1.+xi)
    der(2,7)=-1./4.*(1.+xi)*(xi-2.*eta)
    der(2,8)=-1./2.*(1.-xi**2)
end if
return
end subroutine

subroutine beemat(deriv, bee,nod)
Implicit none
integer, intent(in) :: nod
REAL*8,INTENT(in) :: deriv(:,)
REAL*8,INTENT(out):: bee(:,)
real*8 :: k=0.
integer :: i

Do i=1,nod
    bee(1,(2*i-1))=deriv(1,i)
    bee(1,(2*i))=k
    bee(2,(2*i-1))=k
    bee(2,(2*i))=deriv(2,i)
    bee(3,(2*i-1))=deriv(2,i)
    bee(3,(2*i))=deriv(1,i)
End Do
!bee(1,:)= (/deriv(1,1), k, deriv(1,2), k,deriv(1,3), k, deriv(1,4), k/)
!bee(2,:)= (/k, deriv(2,1),k,deriv(2,2),k,deriv(2,3),k,deriv(2,4)/)
!bee(3,:)= (/deriv(2,1),deriv(1,1),deriv(2,2),deriv(1,2),deriv(2,3),deriv(1,3),deriv(2,4),deriv(1,4)/)
return
end subroutine beemat

subroutine deemat(nst, prop, dee)
Implicit none
integer, intent(in) :: nst
REAL*8,INTENT(in) :: prop(:,)
REAL*8,INTENT(out):: dee(:,)
real*8 :: k=0.0, l=1.0, E, v

E=prop(1,1)
v=prop(2,1)

dee(1,:)=(/l,(v/(1.-v)),k/)
dee(2,:)=(/(v/(1.-v)),l,k/)
dee(3,:)=(/k,k,((1.-2.*v)/(2.*(1.-v))))/)

dee=dee(:,)*(E*(1.-v)/((1.+v)*(1.-2*v)))
return
end subroutine deemat
end module CW3self

```

D Main code

```
PROGRAM plane_strain_elastic_FE
!-----
! Program General two- dimensional analysis of elastic solids in plane
! strain.
!
!-----
USE CW3module ! use library of subroutines and functions
USE CW3self ! use library of subroutines that were created by myself
IMPLICIT NONE
!!*****Initialisation*****!!
!-----fixed variables-----
INTEGER:: &
    i,k,j, & ! i,k, j:simple counter for loops, etc
    iel, & ! counter for elements
    loaded_nodes, & ! number of loaded nodes
    ndim, & ! number of dimensions
    ndof, & ! number of freedoms per element
    nels, & ! number of elements
    neq, & ! total number of (non-zero) freedoms in problem
    nip, & ! number of gauss points
    nn, & ! number of total nodes in problem
    nod, & ! number of nodes per element
    nodof, & ! number of freedoms per node
    nprops=2, & ! number of material properties
    np_types, & ! number of different properties
    nr, & ! number of restrained nodes
    nst, & ! number of stress(strain) terms
    nlen ! number of characters in character string
REAL*8:: &
    det, & ! determinant of the Jacobian matrix
    zero=0.d0
CHARACTER(len=15):: &
    element, & ! types of element
    argv !file name
!-----dynamic arrays-----
INTEGER,ALLOCATABLE:: &
    etype(:,) & ! element property type vector - i.e. vector with which material type each element
    is.
    g(:,) & ! element steering vector - i.e. which freedoms belong to which element
    g_g(:,), & ! global element steering matrix - large vector of g with all elements freedoms
    g_num(:,), & ! global element node numbers matrix
    kdiag(:,) & ! diagonal term location vector
    nf(:,), & ! nodal freedom matrix
    node(:,) & ! loaded nodes vector
    num(:,) ! element node numbers vector
REAL*8,ALLOCATABLE:: &
    bee(:,), & ! strain-displacement matrix
    coord(:,), & ! element nodal coordinates
    dee(:,), & ! stress-strain matrix
    der(:,), & ! shape function derivatives with respect to local coordinates
    deriv(:,), & ! shape function derivatives with respect to global coordinates
    eld(:,) & ! element nodal displacements
```



```

fun(:),    & ! shape functions
gc(:),    & ! integrating point coordinates
g_coord(:,& ! global nodal coordinates
jac(:,& ! Jacobian matrix
km(:,& ! element stiffness matrix
kv(:),    & ! global stiffness matrix
loads(:), & ! nodal loads and displacements
points(:,& ! point local coordinates
prop(:,& ! element properties (E and v for each element)
sigma(:), & ! stress terms
weights(:), & ! weighting coefficients
val_load(:,& ! applied nodal load weightings
coord_l(:,& ! xi and eta of gausspoints
!xi(:)    ! local coordinate xi
!eta(:)   ! local coordinate eta
!-----input and initialisation-----

CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat')
OPEN(11,FILE=argv(1:nlen)//'.res')

READ(10,*)element,nod,nels,nn,nip,nodof,nst,ndim,np_types
ndof=nod*nodof
ALLOCATE(                                     &
  nf(nodof,nn),points(nod,ndim),dee(nst,nst),g_coord(ndim,nn),    &
  coord(nod,ndim),jac(ndim,ndim),weights(nip),num(nod),g_num(nod,nels), &
  der(ndim,nod),deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),eld(ndof), &
  sigma(nst),g(ndof),g_g(ndof,nels),gc(ndim),fun(nod),etype(nels),    &
  prop(nprops,np_types),coord_l(nip,ndim) )
READ(10,*)prop
etype=1
IF (np_types>1) READ(10,*)etype
READ(10,*)g_coord
READ(10,*)g_num                                ! global numbering
!
nf=1
READ(10,*)nr
READ(10,*)(k,nf(:,k),i=1,nr)
CALL formnf(nf)                                ! to set the boundary conditions
neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq))
!
!-----loop over the elements to find global arrays sizes-----
kdiag=0
DO iel=1,nels
  num=g_num(:,iel)
  CALL num_to_g(num,nf,g)
  g_g(:,iel)=g
  CALL fkdiag(kdiag,g)
END DO !elements_1
DO i=2,neq
  kdiag(i)=kdiag(i)+kdiag(i-1)
END DO

```

```

ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')          &
  " There are",neq," equations and the skyline storage is",kdiag(neq)
!
!!*****Caluclate stiffness matrix*****!!
!-----element stiffness integration and assembly-----
!
! <<<  PROGRAM SECTION MISSING --  WRITE ME  >>>
!
call sample(coord_l, points,weights,nod)      !create local xi and eta
kv=0.
Do iel=1,nels          !loop for elements
  coord(:,i)=transpose(g_coord(:,g_num(:,iel))) !match input coordinates with local numbering
  call deemat(nst,prop,dee)    !Create D matrix
  !print*, dee
  km = 0.          !reset km for next element
  deriv = 0.
  do i=1,nip          !loop for all integration points of the element
    call shape_der(coord_l,der,i,nod)!calculate local derivatives of shape functions with xi, eta
    jac=matmul(der,coord)
    !print*, "jacobian", jac
    det=determinant(jac)
    call invert(jac)
    !print*, "ivert jacobian", jac
    deriv=matmul(jac,der)      !calculate global derivatives of shape functions with x and y
    !print*, "deriv", deriv
    call beemat(deriv,bee,nod)  !create B matrix
    !print*, "beemat", bee
    km=km+det*matmul(matmul(transpose(bee),dee),bee)*weights(i)    !assemble stiffness matrix
  km
  !print*, "km", km
end do

!print*, "g_g", g_g(:,1)
call fsparv(kv,km,g_g(:,iel),kdiag) !finds equations that need to be solved and stores them
!in a lower triangular matrix kv
!+assembles it into global stiffness matrix kv
!print*, "kv", kv
!print*, "kdiag", kdiag
end do
!end do
!-----loading condition-----
loads=zero
READ (10,*)loaded_nodes
ALLOCATE (node(loaded_nodes),val_load(loaded_nodes,ndim))
READ (10,*)(node(i),val_load(i,:),i=1,loaded_nodes)
DO i=1,loaded_nodes
  loads(nf(:,node(i)))=val_load(i,:)
END DO
!
!!*****Solve*****!!
!-----equation solution-----
CALL sparin(kv,kdiag)

```

```

CALL spabac(kv,loads,kdiag)
!
!print*, "despl", loads

!!*****Output results*****!!
!-----Output displacements-----

WRITE(11,'(/A)') " Node x-disp y-disp"
DO j=1,nn      !resulting displacements are stored in loads -> write into res file
  WRITE(11,'(I5,2E10.2)')j,loads(nf(:,j))
END DO

!Write(11,*) "Node x-disp y-disp"
!Do j=1,nels      !'for each element'
!  eld=0.          !converting the resulting displacement(stored in 'loads') to eld(:)
!  eld(:)=loads(g_g(:,j))
!  coord(:,:)=transpose(g_coord(:,g_num(:,j)))
!  do i=1,nod      !Writing the displacement in res file
!    k=i+i
!    WRITE(11,'(1I1.0,2E12.4,2E12.4)') g_num(i,j), eld(k-1), eld(k)  !(2E11.4)'
!  end do

!-----recover stresses at element Gauss-points and output-----

WRITE(11,'(/A,I2,A)') " The integration point (nip=",nip,") stresses are:"
WRITE(11,'(A,A)') " Element x-coord y-coord sig_x sig_y tau_xy"

DO iel=1,nels      !loop for all elements
  CALL deemat(nst,prop,dee)      !construct D matrix
  eld=0.
  eld=loads(g_g(:,iel))      !write resulting displacements in eld
  coord=TRANSPOSE(g_coord(:,g_num(:,iel)))!match global numbering of the element with local
numbering
  DO i=1,nip      !loop for all integration points
    CALL shape_fun(coord_l,fun,i,nod) !get N_1-N_i for transofrmation of coordinates of integration
points
    CALL shape_der(coord_l,der,i,nod) !calculate shape-function-derivatives with local coord of
integr points (xi and eta)
    gc=MATMUL(fun,coord)      !calculate global coord of integr points
    jac=MATMUL(der,coord)      !create jacobian
    CALL invert(jac)
    deriv=MATMUL(jac,der)      !create global derivatives of shape functions
    CALL beemat(deriv,bee,nod) !create B matrix
    sigma=MATMUL(dee,MATMUL(bee,eld)) !calculate sigma
    WRITE(11,'(I5,6E10.2)')iel,gc,sigma !plot resulting stresses at integration points
  END DO
END DO

!-----output images-----
CALL mesh(g_coord,g_num,argv,nlen,12)
CALL dismsh(loads,nf,0.05d0,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05d0,0.1d0,g_coord,g_num,argv,nlen,14)
END PROGRAM plane_strain_elastic_FE

```

E Input files

Linear element, 1 element, compression

```
'quadrilateral' 4
1 4 4 2 3 2
1
1.0e6 0.3
0.0 0.0 1.0 0.0 0.0 -1.0 1.0 -1.0
3 1 2 4
4
1 0 1 2 0 1 3 0 0 4 0 0
2
1 0.0 -0.5 2 0.0 -0.5
```

Linear element, 4 elements, compression

```
'quadrilateral' 4
4 10 4 2 3 2
1
1.0e6 0.3
0.0 0.0 1.0 0.0 0.0 -1.0 1.0 -1.0 0.0 -2.0 1.0 -2.0 0.0 -3.0 1.0 -3.0 0.0 -4.0 1.0 -4.0
3 1 2 4 5 3 4 6 7 5 6 8 9 7 8 10
10
1 0 1 2 0 1 3 0 1 4 0 1 5 0 1 6 0 1 7 0 1 8 0 1 9 0 0 10 0 0
2
1 0.0 -0.5 2 0.0 -0.5
```

Linear element, 3 elements, bending

```
'quadrilateral' 4
3 8 4 2 3 2
1
1.0e6 0.3
0.0 0.0 1.0 0.0 0.0 -1.0 1.0 -1.0 0.0 -2.0 1.0 -2.0 0.0 -3.0 1.0 -3.0
3 1 2 4 5 3 4 6 7 5 6 8
8
1 1 1 2 1 1 3 1 1 4 1 1 5 1 1 6 1 1 7 1 0 8 0 0
1
1 1.0 0.0
```

Quadratic elements, 1 element, compression

```
'quadrilateral' 8
1 8 9 2 3 2
1
1.0e6 0.3
0.0 0.0 0.5 0.0 1.0 0.0 0.0 -0.5 1.0 -0.5 0.0 -1.0 0.5 -1.0 1.0 -1.0
6 4 1 2 3 5 8 7
8
1 0 1 2 0 1 3 0 1 4 0 1 5 0 1 6 0 0 7 0 0 8 0 0
3
1 0.0 -0.167 2 0.0 -0.667 3 0.0 -0.167
```

Quadratic elements, 3 elements, bending

```
'quadrilateral' 8
3 18 9 2 3 2
1
1.0e6 0.3
0.0 0.0 0.5 0.0 1.0 0.0 0.0 -0.5 1.0 -0.5 0.0 -1.0 0.5 -1.0 1.0 -1.0 0.0 -1.5 1.0 -1.5 0.0 -2.0
0.5 -2.0 1.0 -2.0 0.0 -2.5 1.0 -2.5 0.0 -3.0 0.5 -3.0 1.0 -3.0
6 4 1 2 3 5 8 7 11 9 6 7 8 10 13 12 16 14 11 12 13 15 18 17
```

```

18
1 1 1 2 1 1 3 1 1 4 1 1 5 1 1 6 1 1 7 1 1 8 1 1 9 1 1 10 1 1 11 1 1 12 1 1 13 1 1 14 1 1 15 1 1 16
0 0 17 0 0 18 0 0
1
1 1.0 0.0

```

F Output files

Linear element, 1 element, compression

There are 2 equations and the skyline storage is 3

Node x-disp y-disp

```

1 0.00E+00 -0.74E-06
2 0.00E+00 -0.74E-06
3 0.00E+00 0.00E+00
4 0.00E+00 0.00E+00

```

The integration point (nip= 4) stresses are:

```

Element x-coord y-coord sig_x sig_y tau_xy
1 0.21E+00 -0.79E+00 -0.43E+00 -0.10E+01 -0.31E-16
1 0.21E+00 -0.21E+00 -0.43E+00 -0.10E+01 -0.12E-15
1 0.79E+00 -0.21E+00 -0.43E+00 -0.10E+01 -0.12E-15
1 0.79E+00 -0.79E+00 -0.43E+00 -0.10E+01 -0.31E-16

```

Linear element, 4 elements, compression

There are 8 equations and the skyline storage is 24

Node x-disp y-disp

```

1 0.0000E+00 -0.2971E-05
2 0.0000E+00 -0.2971E-05
3 0.0000E+00 -0.2229E-05
4 0.0000E+00 -0.2229E-05
5 0.0000E+00 -0.1486E-05
6 0.0000E+00 -0.1486E-05
7 0.0000E+00 -0.7429E-06
8 0.0000E+00 -0.7429E-06
9 0.0000E+00 0.0000E+00
10 0.0000E+00 0.0000E+00

```

The integration point (nip= 4) stresses are:

```

Element x-coord y-coord sig_x sig_y tau_xy
1 0.2113E+00 -0.7887E+00 -0.4286E+00 -0.1000E+01 -0.1629E-15
1 0.2113E+00 -0.2113E+00 -0.4286E+00 -0.1000E+01 -0.6516E-15
1 0.7887E+00 -0.2113E+00 -0.4286E+00 -0.1000E+01 -0.6516E-15
1 0.7887E+00 -0.7887E+00 -0.4286E+00 -0.1000E+01 -0.1629E-15
2 0.2113E+00 -0.1789E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
2 0.2113E+00 -0.1211E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
2 0.7887E+00 -0.1211E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
2 0.7887E+00 -0.1789E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
3 0.2113E+00 -0.2789E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
3 0.2113E+00 -0.2211E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
3 0.7887E+00 -0.2211E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
3 0.7887E+00 -0.2789E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
4 0.2113E+00 -0.3789E+01 -0.4286E+00 -0.1000E+01 0.0000E+00

```

```

4 0.2113E+00 -0.3211E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
4 0.7887E+00 -0.3211E+01 -0.4286E+00 -0.1000E+01 0.0000E+00
4 0.7887E+00 -0.3789E+01 -0.4286E+00 -0.1000E+01 0.0000E+00

```

Linear element, 3 elements, bending

There are 13 equations and the skyline storage is 67

Node x-disp y-disp

```

1 0.70E-04 0.16E-04
2 0.68E-04 -0.15E-04
3 0.37E-04 0.14E-04
4 0.37E-04 -0.14E-04
5 0.12E-04 0.89E-05
6 0.12E-04 -0.84E-05
7 0.12E-05 0.00E+00
8 0.00E+00 0.00E+00

```

The integration point (nip= 4) stresses are:

Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.21E+00	-0.79E+00	0.53E+00	0.16E+01	0.15E+01
1	0.21E+00	-0.21E+00	-0.56E+00	0.11E+01	0.77E+00
1	0.79E+00	-0.21E+00	-0.17E+01	-0.16E+01	0.46E+00
1	0.79E+00	-0.79E+00	-0.62E+00	-0.11E+01	0.12E+01
2	0.21E+00	-0.18E+01	0.19E+01	0.40E+01	0.22E+01
2	0.21E+00	-0.12E+01	0.19E+01	0.40E+01	-0.15E+00
2	0.79E+00	-0.12E+01	-0.15E+01	-0.40E+01	-0.15E+00
2	0.79E+00	-0.18E+01	-0.15E+01	-0.40E+01	0.22E+01
3	0.21E+00	-0.28E+01	0.17E+01	0.65E+01	0.28E+01
3	0.21E+00	-0.22E+01	0.28E+01	0.70E+01	-0.11E+01
3	0.79E+00	-0.22E+01	-0.29E+01	-0.65E+01	-0.77E+00
3	0.79E+00	-0.28E+01	-0.40E+01	-0.70E+01	0.31E+01

Quadratic elements, 1 element, compression

There are 5 equations and the skyline storage is 15

Node x-disp y-disp

```

1 0.0000E+00 -0.7439E-06
2 0.0000E+00 -0.7435E-06
3 0.0000E+00 -0.7439E-06
4 0.0000E+00 -0.3719E-06
5 0.0000E+00 -0.3719E-06
6 0.0000E+00 0.0000E+00
7 0.0000E+00 0.0000E+00
8 0.0000E+00 0.0000E+00

```

The integration point (nip= 9) stresses are:

Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.1127E+00	-0.8873E+00	-0.4291E+00	-0.1001E+01	0.5572E-04
1	0.1127E+00	-0.5000E+00	-0.4291E+00	-0.1001E+01	0.2472E-03
1	0.1127E+00	-0.1127E+00	-0.4291E+00	-0.1001E+01	0.4387E-03
1	0.5000E+00	-0.1127E+00	-0.4289E+00	-0.1001E+01	0.0000E+00
1	0.8873E+00	-0.1127E+00	-0.4291E+00	-0.1001E+01	-0.4387E-03
1	0.8873E+00	-0.5000E+00	-0.4291E+00	-0.1001E+01	-0.2472E-03

```

1 0.8873E+00 -0.8873E+00 -0.4291E+00 -0.1001E+01 -0.5572E-04
1 0.5000E+00 -0.8873E+00 -0.4289E+00 -0.1001E+01 0.0000E+00
1 0.5000E+00 -0.5000E+00 -0.4289E+00 -0.1001E+01 0.2036E-16

```

Quadratic elements, 3 elements, bending

There are 30 equations and the skyline storage is 285

Node x-disp y-disp

```

1 0.10E-03 0.25E-04
2 0.10E-03 -0.78E-07
3 0.99E-04 -0.23E-04
4 0.76E-04 0.23E-04
5 0.76E-04 -0.23E-04
6 0.53E-04 0.21E-04
7 0.52E-04 0.25E-07
8 0.53E-04 -0.21E-04
9 0.33E-04 0.18E-04
10 0.33E-04 -0.18E-04
11 0.16E-04 0.13E-04
12 0.15E-04 -0.33E-08
13 0.16E-04 -0.13E-04
14 0.51E-05 0.65E-05
15 0.51E-05 -0.65E-05
16 0.00E+00 0.00E+00
17 0.00E+00 0.00E+00
18 0.00E+00 0.00E+00

```

The integration point (nip= 9) stresses are:

Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.11E+00	-0.89E+00	0.75E+00	0.41E+01	0.75E+00
1	0.11E+00	-0.50E+00	-0.31E-01	0.29E+01	0.85E+00
1	0.11E+00	-0.11E+00	-0.34E+01	0.68E+00	0.14E+01
1	0.50E+00	-0.11E+00	-0.25E+01	-0.64E+00	0.11E+01
1	0.89E+00	-0.11E+00	-0.91E+00	-0.55E+00	0.80E+00
1	0.89E+00	-0.50E+00	0.78E+00	-0.17E+01	0.90E+00
1	0.89E+00	-0.89E+00	-0.12E+00	-0.40E+01	0.15E+01
1	0.50E+00	-0.89E+00	0.15E-01	-0.64E+00	0.11E+01
1	0.50E+00	-0.50E+00	0.76E-01	-0.80E-01	0.86E+00
2	0.11E+00	-0.19E+01	-0.14E+01	0.86E+01	0.10E+01
2	0.11E+00	-0.15E+01	-0.44E+00	0.70E+01	0.70E+00
2	0.11E+00	-0.11E+01	0.48E+00	0.53E+01	0.11E+01
2	0.50E+00	-0.11E+01	-0.36E-01	0.24E-01	0.13E+01
2	0.89E+00	-0.11E+01	-0.59E+00	-0.53E+01	0.97E+00
2	0.89E+00	-0.15E+01	0.40E+00	-0.70E+01	0.64E+00
2	0.89E+00	-0.19E+01	0.14E+01	-0.87E+01	0.10E+01
2	0.50E+00	-0.19E+01	0.31E-02	0.24E-01	0.13E+01
2	0.50E+00	-0.15E+01	-0.35E-02	0.29E-01	0.96E+00
3	0.11E+00	-0.29E+01	0.51E+01	0.13E+02	0.12E+01
3	0.11E+00	-0.25E+01	0.22E+01	0.12E+02	0.11E+01
3	0.11E+00	-0.21E+01	-0.71E+00	0.99E+01	0.12E+01
3	0.50E+00	-0.21E+01	-0.72E-02	-0.53E-02	0.83E+00
3	0.89E+00	-0.21E+01	0.71E+00	-0.99E+01	0.12E+01
3	0.89E+00	-0.25E+01	-0.22E+01	-0.12E+02	0.11E+01

3 0.89E+00 -0.29E+01 -0.51E+01 -0.13E+02 0.12E+01
3 0.50E+00 -0.29E+01 -0.69E-03 -0.53E-02 0.83E+00
3 0.50E+00 -0.25E+01 0.71E-03 -0.33E-02 0.74E+00