Chapter 12

Finite Element Programming with MATLAB

Chapter 12 includes a general introduction to MATLAB functions, selected topics in linear algebra with MATLAB, and a collection of finite element programs for: trusses (Chapter 2), general one-dimensional problems (Chapter 5), heat conduction in 2D (Chapter 8) and elasticity in 2D (Chapter 9). This Chapter is published electronic format only for several reasons:

1. the data structure of the finite element program will be periodically updated to reflect emerging finite element technologies and MATLAB syntax changes;
2. to allow the course instructors to use their own MALAB or other finite element codes.
3. to create a forum where students and instructors would exchange ideas and place alternative finite element program data structures. The forum is hosted at http://1coursefem.blogspot.com/

12.1 Using MATLAB for FEM[1]

12.1.1 The MATLAB Windows

Upon opening MATLAB you should see three windows: the workspace window, the command window, and the command history window as shown in Figure 12.1. If you do not see these three windows, or see more than three windows you can change the layout by clicking on the following menu selections: View → desktop layout → default.

[1] May not be covered in the class. Recommended as independent reading.

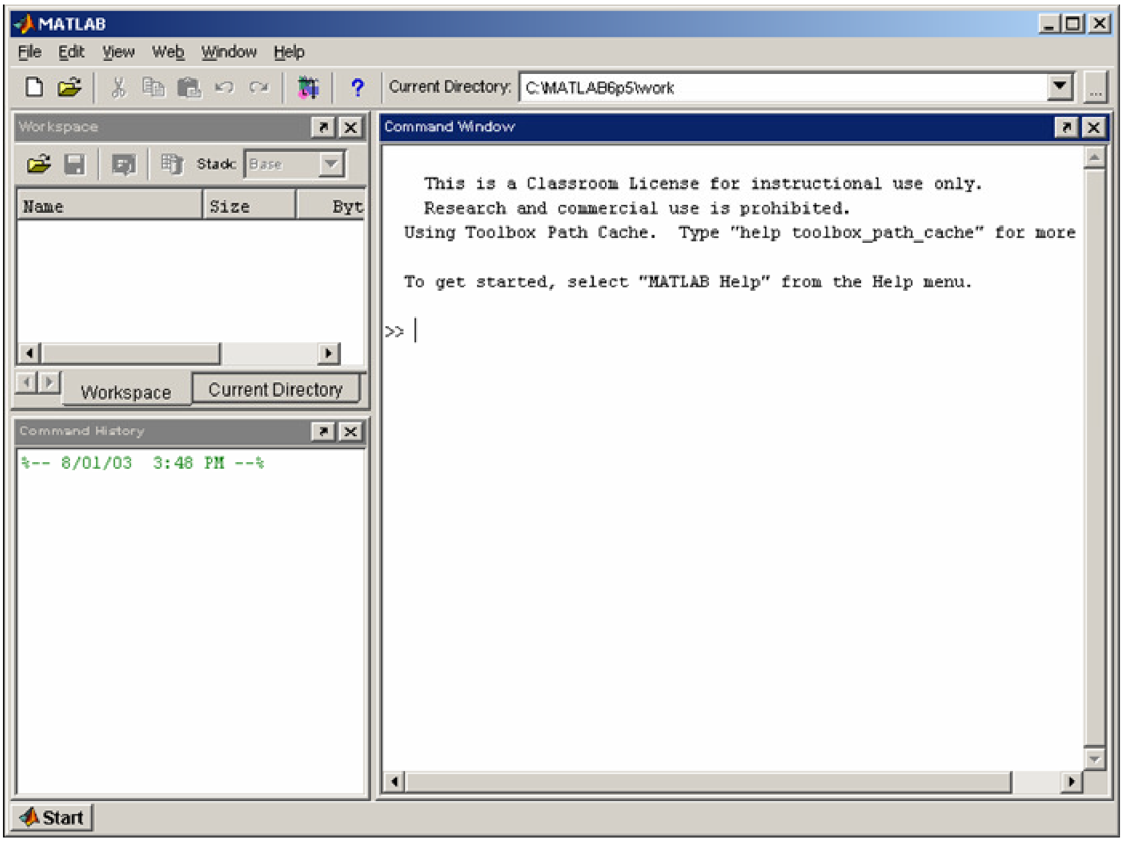

Figure 12.1: Matlab Windows

12.1.2 The Command Window

If you click in the command window a cursor will appear for you to type and enter various commands. The cursor is indicated by two greater than symbols (>>).

12.1.3 Entering Expressions

After clicking in the command window you can enter commands you wish MATLAB to execute. Try entering the following: 8+4. You will see that MATLAB will then return: ans = 12.

12.1.4 Creating Variables

Just as commands are entered in MATLAB, variables are created as well. The general format for entering variables is: variable = expression. For example, enter y = 1 in the command window. MATLAB returns: y = 1. A variable y has been created and assigned a value of 1. This variable can be used instead of the number 1 in future math operations. For example: typing y*y at the command prompt returns: ans = 1. MATLAB is case sensitive, so y=1, and Y=5 will create two separate variables.

12.1.5 Functions

MATLAB has many standard mathematical functions such as sine (sin(x)) and cosine (cos(x)) etc. It also has software packages, called toolboxes, with specialized functions for specific topics.

12.1.6 Getting Help and Finding Functions

The ability to find and implement MATLAB's functions and tools is the most important skill a beginner needs to develop. MATLAB contains many functions besides those described below that may be useful.
There are two different ways obtain help:

- Click on the little question mark icon at the top of the screen. This will open up the help window that has several tabs useful for finding information.

- Type "help" in the command line: MATLAB returns a list of topics for which it has functions. At the bottom of the list it tells you how to get more information about a topic. As an example, if you type "help sqrt" and MATLAB will return a list of functions available for the square root.

12.1.7 Matrix Algebra with MATLAB

MATLAB is an interactive software system for numerical computations and graphics. As the name suggests, MATLAB is especially designed for matrix computations. In addition, it has a variety of graphical and visualization capabilities, and can be extended through programs written in its own programming language. Here, we introduce only some basic procedures so that you can perform essential matrix operations and basic programming needed for understanding and development of the finite element program.

12.1.8 Definition of matrices

A matrix is an *mxn* array of numbers or variables arranged in *m rows* and *n columns*; such a matrix is said to have *dimension mxn* as shown below

$$a = \begin{bmatrix} a_{11} & a_{12} & L & a_{1n} \\ a_{21} & a_{22} & & \\ M & & O & \\ a_{m1} & & & a_{mn} \end{bmatrix}$$

Bold letters will denote matrices or vectors. The elements of a matrix **a** are denoted by a_{ij}, where i is the row number and j is the column number. Note that in both describing the dimension of the matrix and in the subscripts identifying the row and column number, the row number is always placed first.

An example of a 3x3 matrix is:

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

The above matrix **a** is is an example of a *square matrix* since the number of rows and columns are equal.

The following commands show how to enter matrices in MATLAB (>> is the MATLAB prompt; it may be different with different computers or different versions of MATLAB.)

```
>> a = [1 2 3; 4 5 6; 7 8 0]
a =
    1    2    3
    4    5    6
    7    8    0
```

Notice that rows of a matrix are separated by semicolons, while the entries on a row are separated by spaces (or commas). The order of matrix **a** can be determined from

$$size(a)$$

The transpose of any matrix is obtained by interchanging rows and columns. So for example, the *transpose* of **a** is:

$$a^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 0 \end{bmatrix}$$

In MATLAB the transpose of a matrix is denoted by an apostrophe (‘).

If $\mathbf{a}^T = \mathbf{a}$, the matrix $\mathbf{a}$ is *symmetric*.

A matrix is called a column matrix or a vector if *n=1*, e.g.

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

In MATLAB, single subscript matrices are considered row matrices, or row vectors. Therefore, a column vector in MATLAB is defined by

```
>> b = [1  2  3]'
b =
    1
    2
    3
```

Note the transpose that is used to define **b** as a column matrix. The components of the vector **b** are b_1, b_2, b_3. The transpose of **b** is a row vector

$$b^T = [b_1 \quad b_2 \quad b_3]$$

or in MATLAB

```
>> b  = [1  2  3]
b =
     1     2     3
```

A matrix is called a ***diagonal matrix*** if only the diagonal components are nonzero, i.e., $a_{ij} = 0, i \neq j$. For example, the matrix below is a diagonal matrix:

$$\mathbf{a} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

A diagonal matrix in MATLAB is constructed by first defining a row vector b = [1 5 6], and then placing this row vector on the diagonal

```
>> b = [1 5 6];
>> a = diag (b)
a =
     1     0     0
     0     5     0
     0     0     6
```

A diagonal matrix where all diagonal components are equal to one is called an *identity or unit matrix* and is denoted by **I**. For example, 2×2 identity matrix is given by

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The MATLAB expression for an order *n* unit matrix is

$$eye(n)$$

Thus, the MATLAB expression $I = eye(2)$ gives the above matrix.

A matrix in which all components are zero is called a *zero matrix* and is denoted by **0**. In MATLAB, *B* = *zeros (m, n)* creates $m \times n$ matrix *B* of zeros. A random $m \times n$ matrix can be created by *rand (m,n)*.

In finite element method, matrices are often sparse, i.e., they contain many zeros. MATLAB has the ability to store and manipulate sparse matrices, which greatly increases its usefulness for realistic problems. The command *sparse (m, n)* stores an $m \times n$ zero matrix in a sparse format, in which only the nonzero entries and their locations are sorted. The nonzero entries can then be entered one-by-one or in a loop.

```
>> a = sparse (3,2)
a =
   All zero sparse: 3-by-2
>> a(1,2)=1;
>> a(3,1)=4;
>> a(3,2)=-1;
>> a
a =
   (3,1)       4
   (1,2)       1
   (3,2)      -1
```

Notice that the display in any MATLAB statement can be suppressed by ending the line with a semicolon.

The inverse of a square matrix is defined by

$$\mathbf{a}^{-1}\mathbf{a} = \mathbf{a}\mathbf{a}^{-1} = \mathbf{I}$$

if the matrix $\mathbf{a}$ is not singular. The MATLAB expression for the inverse is $inv(a)$. Linear algebraic equations can also be solved by using backslash operator as shown in Section 1.3.10, which avoids computations of the inverse and is therefore faster.

The matrix $\mathbf{a}$ is nonsingular if its determinant, denoted by $\det(\mathbf{a})$, is not equal to zero. A determinant of a 2x2 matrix is defined by

$$a = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \qquad \det(a) = a_{11}a_{22} - a_{12}a_{21}$$

The MATLAB expression for the determinant is

$$\det(a)$$

For example,

```
>> a = [1 3; 4 2];
>> det (a)
ans =
   -10
```

12.1.9 Operation with matrices

Addition and Subtraction

$$c = a \pm b = \begin{bmatrix} a_{11} \pm b_{11} & a_{12} \pm b_{12} & \cdots & a_{1n} \pm b_{1n} \\ a_{21} \pm b_{21} & a_{22} \pm b_{22} & & \\ \vdots & & \ddots & \\ a_{m1} \pm b_{m1} & & & a_{mn} \pm b_{mn} \end{bmatrix}$$

An example of matrix addition in MATLAB is given below:

```
>> a = [1 2 3;4 5 6;7 8 9];
>> a = [1 1 1;2 2 2;3 3 3];
>> c = [1 2;3 4;5 6];
>> a+b
ans =
     2     3     4
     6     7     8
    10    11    12
>> a+c
??? Error using ==> +
Matrix dimensions must agree
```

Multiplication

1. *Multiplication of a matrix by a scalar*

$$c \cdot a = c \cdot \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{m1} & & & a_{mn} \end{bmatrix} = \begin{bmatrix} ca_{11} & ca_{12} & \cdots & ca_{1n} \\ ca_{21} & ca_{22} & & \\ \vdots & & \ddots & \\ ca_{m1} & & & ca_{mn} \end{bmatrix}$$

2. Scalar product of two column vectors

$$a^T \cdot b = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^{n} a_i b_i$$

In MATLAB the scalar product as defined above is given by either $a * b'$ or $dot(a,b)$.

The length of a vector **a** is denoted by |**a**| and is given by

$$\mathbf{a} = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2}$$

The length of a vector is also called its *norm*.

3. Product of two matrices

The product of two matrices **a** $(m \times k)$ and **b** $(k \times n)$ is defined as

$$c = ab = \begin{bmatrix} \sum_{j=1}^{k} a_{1j}b_{j1} & \sum_{j=1}^{k} a_{1j}b_{j2} & \cdots & \sum_{j=1}^{k} a_{1j}b_{jn} \\ \sum_{j=1}^{k} a_{21j}b_{j1} & \sum_{j=1}^{k} a_{2j}b_{j2} & & \\ \vdots & & \ddots & \\ \sum_{j=1}^{k} a_{mj}b_{j1} & & & \sum_{j=1}^{k} a_{mj}b_{jn} \end{bmatrix}$$

Alternatively we can write the above as

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Note the the *i,j* entry of **c** is the scalar product of row *i* of **a** and column *j* of **b**.

The product of two matrices **a** and **b c** is defined only if the number of columns in **a** equals the number of rows in **a**. In other words, if **a** is an $(m \times k)$ matrix, then **b** must be an $(k \times n)$ matrix, where k is arbitrary. The product **c** will then have the same number of rows as **a** and the same number of columns as **b**, i.e. it will be an $m \times n$ matrix.

An important fact to remember is that matrix multiplication is not *commutative,* i.e. $\mathbf{ab} \neq \mathbf{ba}$ except in unusual circumstances.

The MATLAB expression for matrix multiplication is

$$c = a * b$$

Consider the same matrices **a** and **c** as before. An example of matrix multiplication with MATLAB is:

```
>> a*c
ans =
    22    28
    49    64
    76   100
>> c*c
??? Error using ==> *
Inner matrix dimensions must agree.
```

4. *Other matrix operations*

a) Transpose of product: $(\mathbf{ab})^T = \mathbf{b}^T \mathbf{a}^T$
b) Product with identity matrix: $\mathbf{aI} = \mathbf{a}$
c) Product with zero matrix: $\mathbf{a0} = \mathbf{0}$

12.1.10 Solution of system of linear equations

Consider the following system of n equations with n unknowns, d_k, $k = 1, 2, \cdots, n$:

$$\begin{cases} K_{11}d_1 + K_{12}d_2 + \cdots + K_{1n}d_n = f_1 \\ K_{21}d_1 + K_{22}d_2 + \cdots + K_{2n}d_n = f_2 \\ \vdots \\ K_{n1}d_1 + K_{n2}d_2 + \cdots + K_{nn}d_n = f_n \end{cases}$$

We can rewrite this system of equations in matrix notation as follows:

$$\mathbf{Kd} = \mathbf{f}$$

where

$$K = \begin{bmatrix} K_{11} & K_{12} & \cdots & K_{1n} \\ K_{21} & K_{22} & & \\ \vdots & & \ddots & \\ K_{n1} & & & K_{nm} \end{bmatrix} \qquad f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} \qquad d = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}$$

The symbolic solution of the above system of equation can be found by multiplying both sides with inverse of **K**, which yields

$$\mathbf{d} = \mathbf{K}^{-1}\mathbf{f}$$

MATLAB expression for solving the system of equations is

$$d = K \setminus f$$

or

$$d = inv(K) * f$$

An example of solution of system of equations with MATLAB is given below:

```
>> A = rand (3,3)
A =
    0.2190    0.6793    0.5194
    0.0470    0.9347    0.8310
    0.6789    0.3835    0.0346
>> b = rand (3,1)
b =
    0.0535
    0.5297
    0.6711
>> x = A\b
x =
 -159.3380
  314.8625
 -344.5078
```

As mentioned before, the backslash provides a faster way to solve equations and should always be used for large systems. The reason for this is that the backslash uses elimination to solve with one right hand side, whereas determining the inverse of an *n*x*n* matrix involves solving the system with *n* right hand sides. Therefore, the backslash should always be used for solving large system of equations.

12.1.11 Strings in MATLAB

MATLAB variables can also be defined as *string* variables. A string character is a text surrounded by single quotes. For example:

```
>> str='hello world'
str =
hello world
```

It is also possible to create a list of strings by creating a matrix in which each row is a separate string. As with all standard matrices, the rows must be of the same length. Thus:

```
>> str_mat = ['string A' ; 'string B']
str_mat =
string A
string B
```

Strings are used for defining file names, plot titles, and data formats. Special built-in string manipulation functions are available in MATLAB that allow you to work with strings. In the MATALB codes provided in the book we make use of strings to compare functions. For example the function *strcmpi* compares two strings

```
>> str = 'print output';
>> strcmpi(str,'PRINT OUTPUT')
ans =
    1
```

A true statment results in *1* and a false statement in *0*. To get a list of all the built-in MATLAB functions type

```
>> help strfun
```

Another function used in the codes is *fprintf*. This function allows the user to print to the screen (or to a file) strings and numeric information in a tabulated fasion. For example

```
>>fprintf(1,'The number of nodes in the mesh is %d \n',10)
The number of nodes in the mesh is 10
```

The first argument to the function tells MATLAB to print the message to the screen. The second argument is a string, where *%d* defines a decimal character with the value of *10* and the *\n* defines a new line. To get a complete description type

>> help fprintf

12.1.11 Programming with MATLAB

MATLAB is very convenient for writing simple finite element programs. It provides the standard constructs, such as loops and conditionals; these constructs can be used interactively to reduce the tedium of repetitive tasks, or collected in programs stored in "m-files" (nothing more than a text file with extension ``.m").

12.1.11.1 Conditional and Loops

MATLAB has a standard if-elseif-else conditional.

The general form	An example

if expression1
  statements1
elseif expression2
  statements2
…
…
…
else
  statements
end
	>> t = 0.76;
	>> if t > 0.75
	    s = 0;
	  elseif t < 0.25
	    s = 1;
	  else
	    s = 1-2*(t-0.25);
	  end
	>> s
	s =
	    0

MATLAB provides two types of loops, a for-loop (comparable to a Fortran do-loop or a C for-loop) and a while-loop. A for-loop repeats the statements in the loop as the loop index takes on the values in a given row vector; the while-loop repeats as long as the given expression is true (nonzero):

The general form	Examples

for index = start:increment:end
  statements
end
	>> for i=1:1:3
	    disp(i^2)
	  end
	   1
	   4
	   9

while expression
  statements
end
	>> x=1;
	>> while 1+x > 1
	    x = x/2;
	  end

```
>> x
x =
   1.1102e-16
```

12.1.11.2 Functions

Functions allow the user to create new MATLAB commands. A function is defined in an m-file that begins with a line of the following form:

function [output1,output2,...] = cmd_name(input1,input2,...)

The rest of the m-file consists of ordinary MATLAB commands computing the values of the outputs and performing other desired actions. Below is a simple example of a function that computes the quadratic function $f(x) = x^2 - 3x - 1$. The following commands should be stored in the file *fcn.m* (the name of the function within MATLAB is the name of the m-file, without the extension)

```
function y = fcn( x )
y=x^2-3*x-1;
Then type command:
>> fcn(0.1)
ans =
   -1.2900
```

12.1.12 Basic graphics

MATLAB is an excellent tool for visualizing and plotting results. To plot a graph the user specifies the x coordinate vector and y coordinate vector using the following syntax

```
>> x=[0:0.01:1];
>> y=x.^2;
>> plot(x,y);
```

The above will generate

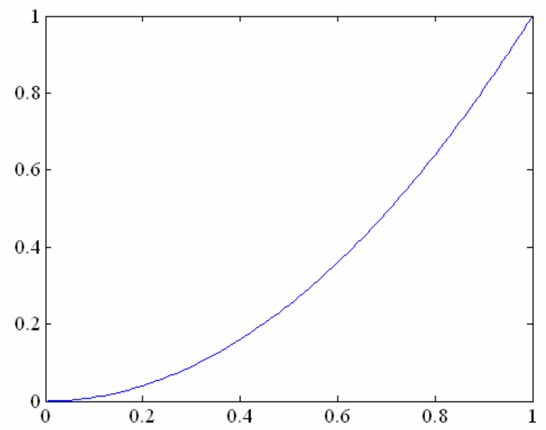

Figure 12.2 Typical outpout of *plot(x,y)* function

Various line types, plot symbols and colors may be obtained with plot*(x,y,s)* where s is a character string consisting of elements from any combination of the following 3 columns:

b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		

To add a title, x and y labels, or a grid, the user should use the following MATLAB functions. Note that the arguments to the functions are strings

```
>> title('circle');
>> xlabel('x');
>> ylabel('y');
>> grid
```

In the MATLAB Finite Element code provided in the book, we also use two specialized plots. The first plot is the *patch* function. This function is used to visualize 2D polygons with colors. The colors are interpolated from nodes of the polygon to create a colored surface. The following example generates a filled square. The colors along the x axis are the same while the colors along the y axis are interpolated between the values [0,1].

```
>> x = [0 1 1 0];
>> y = [0 0 1 1];
>> c = [0 0 1 1];
>> patch(x,y,c)
```

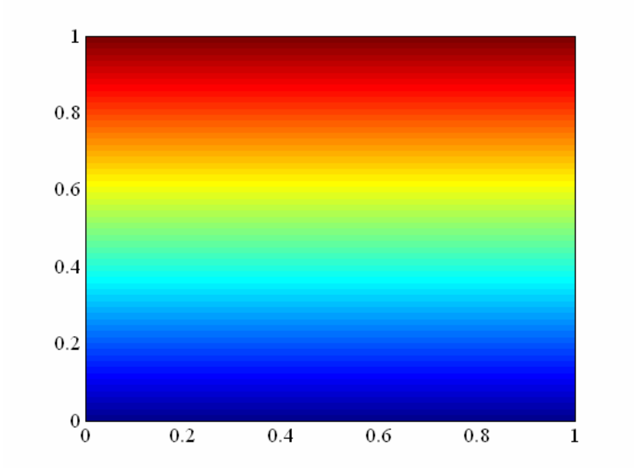

Figure 12.3 Typical outpout of *patch(x,y,c)* function

We will use the *patch* function to visualize temperatures, stresses and other variables obtained at the finite element solutions. Another specialized plot function is the *quiver*. This function is used to visualize gradients of functions as an arrow plot. The following example demonstrates the use of *quiver* function for plotting the gradients to the function $y=x^2$

```
>> x=0:0.1:1; y=x.^2;
>> cx=ones(1,11); cy=2*x;
>> plot(x,y); hold on
>> quiver(x,y,cx,cy)
```

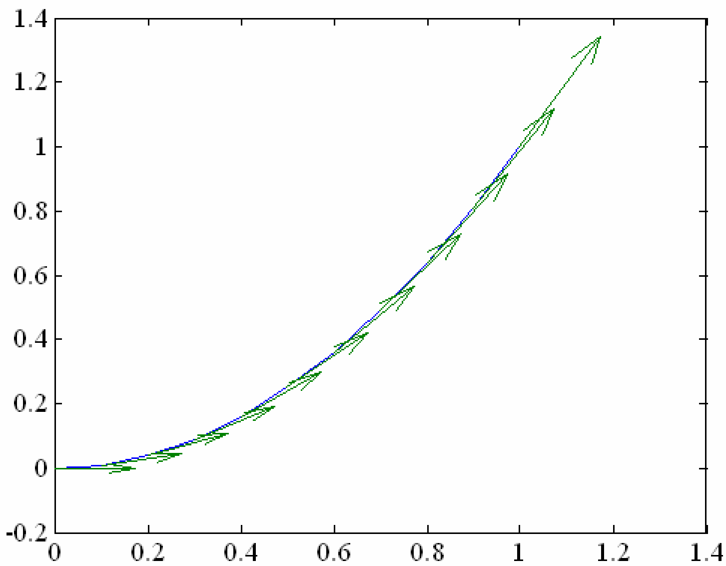

Figure 12.4 Typical outpout of *quiver(x,y,cx,cy)* function

The *hold on* command is used to hold the current plot and all axis properties so that subsequent graphing commands will executed on the existing graph.
Using the *text* function, the user can add to a plot a text message. For example

text(1,1,'flux')

The first and second arguments define the position of the text on the plot, while the string gives the text.

12.1.13 Remarks

a) In practice the number of equations *n* can be very large. PCs can today solve thousands of equations in a matter of minutes if they are sparse (as they are in FEM analysis-you will learn about this later) but sometimes millions of equations are needed, as for an aircraft carrier or a full model of an aircraft; parallel computers are then needed.
b) Efficient solution techniques that take advantage of the sparsity and other advantageous properties of FEM equations are essential for treating even moderately large systems. The issue of how to efficiently solve large systems will not be considered in this course.
c) In this course, we will see that
   - The matrix corresponding to the system of equations arising from FEM (denoted as **K**) is non-singular (often called regular), i.e., $\mathbf{K}^{-1}$ exists if the correct boundary conditions are prescribed and the elements are properly formulated. Furthermore, for good models it is usually well-conditioned, which means it is not very sensitive to roundoff errors.

- $\mathbf{K}$ is symmetric, i.e. $\mathbf{K}^T = \mathbf{K}$.
- $\mathbf{K}$ is positive definite, i.e., $\mathbf{x}^T\mathbf{K}\mathbf{x} > 0$ " $\mathbf{x}$ (meaning for any value of $\mathbf{x}$) Alternatively, $\mathbf{K}$ is said to be positive definite if all the eigenvalues are strictly positive. The eigenvalue problem consists of finding nonzero eigenvectors $\mathbf{y}$ and the corresponding eigenvalues l satisfying

$$\mathbf{K}\mathbf{y} = l\,\mathbf{y}$$

The MATLAB expression for the eigenvalues problem is:

```
>> K=[2 -2;-2 4];
>> [y, lamda]=eig(K)
y =
    0.8507   -0.5257
   -0.5257    0.8507
lamda =
    0.7639         0
         0    5.2361
```

12.2 Finite element programming with MATLAB for trusses

In Chapter 2 the basic structure of the finite element method for truss structures has been illustrated. In this section we present a simple finite element program using MATLAB programming language. Since MATLAB manipulates matrices and vectors with relative ease the reader can focus on fundamentals ideas rather than on algorithmic details.

The code is written to very closely follow the formulation given in this chapter. In order to better understand how the program works Figure 2.8 and Example Problem 2.2 in Chapter 2 have been included as examples solved by the program. Going through the code along with this guide and the example problems is an effective method to comprehend the program.

The main routines in the finite element code are:

1. Preprocessing including input data and assembling the proper arrays, vectors, and matrices.
2. Calculation of element stiffness matrices and force vectors
3. Direct assembly of matrices and vectors
4. Partition and solution
5. Postprocessing for secondary variables

Explanation for various MATLAB routines (stored in *.m files) are described as comments within each subroutine.

12.2.1 Notations and definitions

12.2.1.1 User provided

nsd: number of space dimension (1 for 1D problems)
ndof: number of degrees-of-freedom per node
nnp: number of nodal points
nel: number of elements
nen: number of element nodes (2 in this case)
nd: number of prescribed (known) displacements
CArea: cross-sectional area
Area = **CArea***(element number)*
E: Young's Modulus
Young = **E***(element number)*
leng: element length
Length = **leng***(element number)*
phi: angle from $x¢$axis to x axis for each element specified in degrees. Remember, $x¢$ is
always from local node 1 to 2
phi = **phi***(element number)*
IEN: connectivity information matrix
global node number = **IEN** *(local node number, element number)*
d_bar: prescribed displacement vector **-** $\bar{\mathbf{d}}$ in Eq. **Error! Reference source not found.**.
f_hat: given force vector **-** $\hat{\mathbf{f}}$ in Eq. **Error! Reference source not found.**.

plot_truss: string for output control: ['yes'] to plot truss elements
plot_nod: string for output control: ['yes'] to plot truss global node numbers
plot_stress: string for output control: ['yes'] to plot stresses

12.1.1.2 Calculated or derived by program

neq: total number of equations
K: global stiffness matrix
d: global displacement vector is stored as:

for 1-D problems

$$\begin{bmatrix} u_1 \\ M \\ u_n \end{bmatrix} = d$$

for 2-D problems

$$\begin{bmatrix} u_{1x} \\ u_{1y} \\ M \\ u_{ny} \end{bmatrix} = d$$

f: global force vector (excluding the reactions) is stored as:

for 1-D problems

$$\begin{bmatrix} f_1 \\ M \\ f_n \end{bmatrix} = f$$

for 2-D problems

$$\begin{bmatrix} f_{1x} \\ f_{1y} \\ M \\ f_{ny} \end{bmatrix} = f$$

e: element number
ke: element stiffness matrix
de: element nodal displacement vector:

for 1-D problems

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = de$$

for 2-D problems

$$\begin{bmatrix} u_{1x}^e \\ u_{1y}^e \\ u_{2x}^e \\ u_{2y}^e \end{bmatrix} = de$$

LM: gather matrix
The gather matrix is used to extract the element and local degrees-of-freedom. It has the following structure:

global degree-of-freedom=**LM** *(local degree-of-freedom, element number)*

When **ndof = 1** (see example in Figure 2.8) IEN and LM are defined as follows:

$$\begin{array}{c} \begin{array}{cc} e{=}1 & e{=}2 \end{array} \\ \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \end{array} = IEN \qquad \begin{array}{c} \begin{array}{cc} e{=}1 & e{=}2 \end{array} \\ \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \end{array} = LM$$

When **ndof = 2** (example Problem 2.2), IEN and LM are defined as:

$$\begin{array}{c} \begin{array}{cc} e{=}1 & e{=}2 \end{array} \\ \begin{bmatrix} 1 & 2 \\ 3 & 3 \end{bmatrix} \end{array} = IEN \qquad \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 5 & 5 \\ 6 & 6 \end{bmatrix} = LM$$

In both examples, columns indicate the elements and rows indicate global degrees-of-freedom.

K_E: partition of the global stiffness matrix **K** based on Eq. **Error! Reference source not found.**
K_EF: partition of the global stiffness matrix **K** based on Eq. **Error! Reference source not found.**
K_F: partition of the global stiffness matrix **K** based on Eq. **Error! Reference source not found.**
d_F: unknown (free) part of the global displacement vector **d** based on Eq. **Error! Reference source not found.**

d_E: prescribed (essential) part of the global displacement vector **d** based on Eq. **Error! Reference source not found.**

f_E: reaction force (unknown) vector based on Eq. **Error! Reference source not found.**

stress: stress for each element

Remark: In this chapter nodes where the displacements are prescribed have to be numbered first.

12.21.2 MATLAB Finite element code for trusses

truss.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 2D Truss (Chapter 2)                    %
% Haim Waisman, Rensselaer                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;
close all;

% include global variables
include_flags;

% Preprocessor Phase
 [K,f,d]  = preprocessor;

% Calculation and assembly of element matrices
for e = 1:nel
    ke     = trusselem(e);
    K      = assembly(K,e,ke);
end

% Solution Phase
 [d,f_E] = solvedr(K,f,d);

% Postprocessor Phase
postprocessor(d)
```

include_flags.m

```
% file to include global variables
global   nsd ndof nnp nel nen neq nd
global   CArea E leng phi
global   plot_truss plot_nod plot_stress
global   LM IEN x y stress
```

preprocessor.m

```
% preprocessing– read input data and set up mesh information
function  [K,f,d]  = preprocessor;
include_flags;
```

```
% input file to include all variables
input_file_example2_2;
%input_file_example2_8;

% generate LM array
for e = 1:nel
   for j = 1:nen
      for m = 1:ndof
         ind = (j-1)*ndof + m;
         LM(ind,e) = ndof*IEN(j,e) - ndof + m;
      end
   end
end
```

input_file_example2_2.m

```
% Input Data for Example 2.2
nsd      = 2;      % Number of space dimensions
ndof     = 2;      % Number of degrees-of-freedom per node
nnp      = 3;      % Number of nodal points
nel      = 2;      % Number of elements
nen      = 2;      % Number of element nodes

neq      = ndof*nnp;      % Number of equations

f        = zeros(neq,1);  % Initialize force vector
d        = zeros(neq,1);  % Initialize displacement matrix
K        = zeros(neq);    % Initialize stiffness matrix

% Element properties
CArea    = [1      1   ];    % Elements area
leng     = [1   sqrt(2)];    % Elements length
phi      = [90     45  ];    % Angle
E        = [1      1   ];    % Young's Modulus

% prescribed displacements
% displacement     d1x    d1y    d2x    d2y
d        = [0      0      0      0]';
nd       = 4;      % Number of prescribed displacement degrees-of-freedom

% prescribed forces
f(5)     = 10;     % Force at node 3 in the x-direction
f(6)     = 0;      % Force at node 3 in the y-direction

% output plots
plot_truss        = 'yes';
plot_nod          = 'yes';

% mesh Generation
truss_mesh_2_2;
```

truss_mesh_2_2.m

```
% geometry and connectivity for example 2.2
function  truss_mesh_2_2
include_flags;

% Nodal coordinates   (origin placed at node 2)
x          =  [1.0  0.0  1.0  ];     % x coordinate
y          =  [0.0  0.0  1.0  ];     % y coordinate

% connectivity array
IEN =  [1    2
           3    3];

% plot truss
plottruss;
```

input_file_example2_8.m

```
% Input Data from Chapter 2 Figure 2.8
nsd       = 1;    % Number of spatial dimensions
ndof      = 1;    % Number of degrees-of-freedom per node
nnp       = 3;    % Total number of global nodes
nel       = 2;    % Total number of elements
nen       = 2;    % Number of nodes in each element

neq       = ndof*nnp;        % Number of equations

f         = zeros(neq,1);  % Initialize force vector
d         = zeros(neq,1);  % Initialize displacement vector
K         = zeros(neq);      % Initialize stiffness matrix

% Element properties
CArea   = [.5        1];         % Elements cross-sectional area
leng     = [2         2];         % Elements length
E          = [1         1];         % Young's Modulus

% prescribed displacements
d(1)      = 0;
nd         = 1;                     % Number of prescribed displacement degrees of freedom

% prescribed forces
f(3)       = 10;                   % force at node 3 in the x-direction

% output controls
plot_truss  = 'yes';
plot_nod    = 'yes';

% mesh generation
truss_mesh_2_8;
```

truss_mesh_2_8.m

```
% geometry and connectivity for example problem in Figure 2.8
function  truss_mesh_2_8;
include_flags;

% Node coordinates   (origin placed at node 1)
x          =  [0.0  1.0  2.0  ];     % x coordinate
y          =  [0.0  0.0  0.0  ];     % y coordinate

% connectivity array
IEN =  [1    2
          2    3];

% plot truss
plottruss;
```

Plottruss.m

```
% function to plot the elements, global node numbers and print mesh parameters
function plottruss;
include_flags;

% check if  truss plot is requested
if strcmpi(plot_truss,'yes')==1;
   for i = 1:nel
      XX = [x(IEN(1,i)) x(IEN(2,i)) x(IEN(1,i)) ];
      YY = [y(IEN(1,i)) y(IEN(2,i)) y(IEN(1,i)) ];
      line(XX,YY);hold on;

      % check if node numbering is requested
      if strcmpi(plot_nod,'yes')==1;
         text(XX(1),YY(1),sprintf('%0.5g',IEN(1,i)));
         text(XX(2),YY(2),sprintf('%0.5g',IEN(2,i)));
      end
   end
   title('Truss Plot');
end

% print mesh parameters
fprintf(1,'\tTruss Params \n');
fprintf(1,'No. of Elements  %d \n',nel);
fprintf(1,'No. of Nodes     %d \n',nnp);
fprintf(1,'No. of Equations %d \n\n',neq);
```

trusselem.m

```
% generate the element stiffness matrix  for each element
function ke = trusselem(e)
include_flags;

const = CArea(e)*E(e)/leng(e);    % constant coefficient within the truss element
```

```
if ndof == 1
    ke = const * [1   -1 ;        % 1-D stiffness
                  -1    1];
elseif ndof == 2
    p = phi(e)*pi/180;       % Converts degrees to radians

    s   = sin(p);        c  = cos(p);
    s2  = s^2;           c2 = c^2;

    ke = const*[c2     c*s    -c2    -c*s;       % 2-D stiffness
                c*s    s2     -c*s   -s2;
                -c2    -c*s   c2     c*s;
                -c*s   -s2    c*s    s2];

end
```

assembly.m

```
% assemble element stiffness matrix
function K = assembly(K,e,ke)
include_flags;

for loop1 = 1:nen*ndof
    i = LM(loop1,e);
    for loop2 =  1:nen*ndof
        j = LM(loop2,e);
        K(i,j) = K(i,j) + ke(loop1,loop2);
    end
end
```

solvedr.m

```
% partition and solve the system of equations
function [d,f_E] = solvedr(K,f,d)
include_flags;

% partition the matrix K, vectors f and d
K_E      = K(1:nd,1:nd);                    % Extract K_E matrix
K_F      = K(nd+1:neq,nd+1:neq);            % Extract K_E matrix
K_EF     = K(1:nd,nd+1:neq);                % Extract K_EF matrix
f_F      = f(nd+1:neq);                     % Extract f_F vector
d_E      = d(1:nd);                         % Extract d_E vector

% solve for d_F
d_F      =K_F\( f_F - K_EF'* d_E);

% reconstruct the global displacement d
d = [d_E
     d_F];

% compute the reaction r
```

```
f_E = K_E*d_E+K_EF*d_F;

% write to the workspace
solution_vector_d          = d
reactions_vector           = f_E
```

postprocessor.m

```
% postprocessing function
function postprocesser(d)
include_flags;

% prints the element numbers and  corresponding stresses
   fprintf(1,'element\t\t\tstress\n');
   % compute stress vector
   for e=1:nel
      de           = d(LM(:,e));      % displacement at the current element
      const        = E(e)/leng(e);  % constant parameter within the element

      if ndof == 1     % For 1-D truss element
         stress(e) = const*([-1 1]*de);
      end
      if ndof == 2     % For 2-D truss element
         p = phi(e)*pi/180;                         % Converts degrees to radians
         c = cos(p);    s = sin(p);
         stress(e) = const*[-c -s c s]*de;             % compute stresses
      end

      fprintf(1,'%d\t\t\t%f\n',e,stress(e));
   end
```

12.3 Shape functions and Gauss quadrature with MATLAB

In Chapter 2 the basic finite element programming structure was introduced for one- and two-dimensional analysis of truss structures. In this section we give the functions for the construction of element shape functions in one-dimension and their derivatives. The shape functions are defined in the physical coordinate system.

12.3.1 Notations and definitions

xe: element nodal x-coordinates
xt: x coordinate at which the functions are evaluated
N: array of shape functions
B: array of derivatives of the shape functions
gp: array of position of Gauss points in the parent element domain - $\lfloor x_1 \quad x_2 \quad L \quad x_{ngp} \rfloor$
W: array of weights - $\lfloor W_1 \quad W_2 \quad L \quad W_{ngp} \rfloor$

12.3.2 MATLAB code for shape functions and derivatives

Nmatrix1D.m

```
% shape functions computed in the physical coordinate - xt
 function N = Nmatrix1D(xt,xe)
include_flags;

   if nen == 2         % linear shape functions
      N(1) = (xt-xe(2))/(xe(1)-xe(2));
      N(2) = (xt-xe(1))/(xe(2)-xe(1));
   elseif nen == 3     % quadratic shape functions
      N(1)=(xt-xe(2))*(xt-xe(3))/((xe(1)-xe(2))*(xe(1)-xe(3)));
      N(2)=(xt-xe(1))*(xt-xe(3))/((xe(2)-xe(1))*(xe(2)-xe(3)));
      N(3)=(xt-xe(1))*(xt-xe(2))/((xe(3)-xe(1))*(xe(3)-xe(2)));
   end
```

Bmatrix1D.m

```
% derivative of the shape functions computed in the physical coordinate - xt
 function B = Bmatrix1D(xt,xe)
include_flags;

   if nen == 2       % derivative of linear shape functions (constant)
      B =  1/(xe(1)-xe(2))*[-1 1];
   elseif nen == 3   % derivative of quadratic shape functions
      B(1)=(2*xt-xe(2)-xe(3))/((xe(1)-xe(2))*(xe(1)-xe(3)));
      B(2)=(2*xt-xe(1)-xe(3))/((xe(2)-xe(1))*(xe(2)-xe(3)));
      B(3)=(2*xt-xe(1)-xe(2))/((xe(3)-xe(1))*(xe(3)-xe(2)));
   end
```

12.3.3 MATLAB code for Gauss quadrature

gauss.m

```
% get gauss points in the parent element domain [-1, 1] and the corresponding weights
function [w,gp] = gauss(ngp)

   if ngp == 1
      gp = 0;
      w  = 2;
   elseif ngp == 2
      gp = [-0.57735027, 0.57735027];
      w  = [1,           1];
   elseif ngp == 3
      gp = [-0.7745966692,  0.7745966692,  0.0];
      w  = [0.5555555556,   0.5555555556,  0.8888888889];
   end
```

12.4 Finite element programming in 1D with MATLAB

In Section 12.2 the basic finite element programming structure was introduced for one- and two- dimensional analysis of truss structures. In 12.3, the program functions for the calculation of the element shape functions, their derivatives and Gauss quadrature in one-dimension were introduced. In this section we introduce a more general finite element program structure for one-dimensional problems that in principle is similar to that in multidimensions to be developed in Sections 12.5 and 12.6 for heat conduction and elasticity problems, respectively.

In Chapter 2 we discussed various methodologies for imposing boundary conditions. In the partition-based approach, the so-called *E*-nodes (where displacements are prescribed) are numbered first. In general, however, node and element numberings are initially defined by mesh generators and subsequently renumbered to maximize efficiency of solving a system of linear equations. In our implementation we tag nodes located on the natural boundary or essential boundary. Nodes on a natural boundary are assigned ***flag=1***, while nodes on an essential boundary are tagged as ***flag=2***. Subsequently, nodes are renumbered by the program so that *E*-nodes are numbered first. This is accomplished by constructing the ***ID*** and ***LM*** arrays in the function ***setup_ID_LM***. With some minor modifications the program for the one-dimensional elasticity problems can be modified to analyze heat conduction problems.

Explanation for various MATLAB routines is given as comments within each function.

Only the nomenclature and definitions which have been modified from the previous chapters are included below. Much of the code is either identical or very similar to the code developed in Section 12.2. An input file for the Example 5.2 in Chapter 5 modeled with two quadratic elements is given below. Additional input files for one quadratic element mesh and four quadratic elements mesh are provided in the disk.

12.4.1 Notations and definitions

User provided

nd: number of nodes on the essential boundary (*E*-nodes)
ngp: number of Gauss points
body: vector of values of body forces – defined at the nodes and then interpolated using shape functions
E: vector of nodal values of Young's modulus
CArea: vector of nodal values of cross-sectional area
flags: Flag array denoting essential and natural boundary conditions

flags(Initial global node number) = flag value
Flag values are: 1 – natural boundary; 2 – essential boundary

x: vector of nodal *x*-coordinates
y: vector of nodal *y*-coordinates (used for the plots only)
e_bc: vector of essential boundary conditions (displacements or temperatures)
n_bc: vector of natural boundary conditions (tractions or boundary fluxes)
P: vector of point forces (point sources in heat conduction)
xp: vector of the *x*-coordinates where the point forces are applied
np: number of point forces (point sources in heat conduction)
nplot: number of points used to plot displacements and stresses (temperatures and fluxes in heat conduction)
IEN: location matrix
The location matrix relates initial global node number and element local node numbers. Subsequently nodes are renumbered (see *setup_ID_LM.m*) so that *E*-nodes are numbered first. IEN matrix has the following structure:

$$\boxed{Initial\ global\ node\ number\ = IEN\ (local\ node\ number,\ element\ number)}$$

Calculated by FE program:

ID: Destination array

$$\boxed{Reordered\ global\ node\ number\ = ID\ (Initial\ global\ node\ number)}$$

LM: Location matrix

$$\boxed{Reordered\ global\ node\ number\ = LM\ (Local\ node\ number,\ element\ number)}$$

Note that *LM* matrix is related to *IEN* matrix by

$$\boxed{LM(I,e) = ID(IEN(I,e))}$$

12.4.2 MATLAB Finite element code for one-dimensional problems

bar1D.m

```
%%%%%%%%%%%%%%%%%%%%%
% 1D FEM Program (Chapter 5)    %
% Haim Waisman, Rensselaer      %
%%%%%%%%%%%%%%%%%%%%%%%
clear all;
close all;

% include global variables
include_flags;

% Preprocessing
 [K,f,d] = preprocessor;

% Element matrix computations and assembly
for e = 1:nel
```

```
   [ke,fe] = barelem(e);
   [K, f]  = assembly(K,f,e,ke,fe);
end

% Add nodal boundary force vector
f  = NaturalBC(f);

% Partition and solution
[d,f_E] = solvedr(K,f,d);

% Postprocessing
postprocessor(d);

% plot the exact solution
ExactSolution;
```

include_flags.m

```
% Include global variables
global nsd ndof nnp nel nen neq nd CArea E
global flags ID IEN LM body x y
global xp P ngp xplot n_bc e_bc np
global plot_bar plot_nod nplot
```

preprocessor.m

```
% preprocessing– reads input data and sets up mesh information
function  [K,f,d] = preprocessor;
include_flags;

% input file to include all variables
input_file5_2_2ele;
%input_file5_2_1ele;
%input_file5_2_4ele;

% generate LM and ID arrays
d = setup_ID_LM(d);
```

input_file5_2_2ele.m

```
% Input Data for Example 5.2 (2 elements)

nsd      = 1;      % number of space dimensions
ndof     = 1;      % number of degrees-of-freedom per node
nnp      = 5;      % number of nodal points
nel      = 2;      % number of elements
nen      = 3;      % number of element nodes

neq = ndof*nnp;            % number of equations

f          = zeros(neq,1);  % initialize nodal force vector
```

```
d          = zeros(neq,1);  % initialize nodal displacement vector
K          = zeros(neq);    % initialize stiffness matrix

flags      = zeros(neq,1); % initialize flag vector
e_bc       = zeros(neq,1); % initialize vector of essential boundary condition
n_bc       = zeros(neq,1);  % initialize vector of natural boundary condition

% element and material data (given at the element nodes)
E          =  8*ones(nnp,1);               % nodal values Young's modulus
body       =  8*ones(nnp,1);               % nodal values body forces
CArea      =  [4     7    10    11    12]';    % nodal values of cross-sectional area

% gauss integration
ngp       = 2;         % number of gauss points

% essential boundary conditions
flags(1) = 2;       % flags to mark nodes located on the essential boundary
e_bc(1) = 0;        % value of essential B.C
nd         = 1;        % number of nodes on the essential boundary

% natural boundary conditions
flags(5) = 1;       % flags to mark nodes located on the natural boundary
n_bc(5) = 0;        % value of natural B.C

% point forces
P          = 24;       % array of point forces
xp         = 5;        % array of  coordinates where point forces are applied
np         = 1;        % number of point forces

% output plots
plot_bar             = 'yes';
plot_nod             = 'yes';
nplot      = nnp*10;    % number of points in the element to plot displacements and stresses

 % mesh generation
bar_mesh5_2_2ele;
```

bar_mesh5_2_2ele.m

```
function  bar_mesh5_2_2ele
include_flags;

% Node:  1    2    3    4    5
x   =  [2.0  3.5  5.0  5.5  6.0  ];        % x coordinate
y   =  2*x;                                 % y is used only for the bar plot

% connectivity array
IEN        = [ 1    3
               2    4
               3    5];
plotbar;
```

setup_ID_LM.m

```
% setup ID and LM arrays
function  d = setup_ID_LM(d);
include_flags;

count = 0; count1 = 0;
for i = 1:neq
    if flags(i) == 2            % check if essential boundary
        count   = count + 1;
        ID(i)   = count;        % number first the nodes on essential boundary
        d(count)= e_bc(i);      % store the reordered values of essential B.C
    else
        count1 = count1 + 1;
        ID(i) = nd + count1;
    end
end

for i = 1:nel
    for j = 1:nen
        LM(j,i)=ID(IEN(j,i));   % create the LM matrix
    end
end
```

barelem.m

```
% generate element stiffness matrix and element nodal body force vector
function [ke, fe] = barelem(e);
include_flags;

IENe     = IEN(:,e);              % extract local connectivity information
xe       = x(IENe);               % extract element x coordinates
J        = (xe(nen) - xe(1))/2;   % compute Jacobian
[w , gp] = gauss(ngp);            % extract Gauss points and weights

ke       = zeros(nen,nen);        % initialize element stiffness matrix
fe       = zeros(nen,1);          % initialize element nodal force vector

for i = 1:ngp
  xt     = 0.5*(xe(1)+xe(nen))+J*gp(i);   % Compute Gauss points in physical coordinates

  N      = Nmatrix1D(xt,xe);      % shape functions matrix
  B      = Bmatrix1D(xt,xe);      % derivative of shape functions matrix

  Ae     = N*CArea(IENe);         % cross-sectional area at element gauss points
  Ee     = N*E(IENe);             % Young's modulus at element gauss points
  be     = N*body(IENe);          % body forces at element gauss points
  ke     = ke + w(i)*(B'*Ae*Ee*B);        % compute element stiffness matrix
  fe     = fe  + w(i)*N'*be;              % compute element nodal body force vector
end
ke       = J*ke;
fe       = J*fe;
```

```
% check for point forces in this element
for i=1:np                      % loop over all point forces
    Pi      = P(i);             % extract point force
    xpi     = xp(i);            % extract the location of point force within an element
    if xe(1)<=xpi &  xpi<xe(nen)
        fe  = fe + Pi*[Nmatrix1D(xpi,xe)]';     % add to the nodal force vector
    end
end
```

assembly.m

```
% assemble element stiffness matrix and nodal force vector
function [K,f] = assembly(K,f,e,ke,fe)
include_flags;

for loop1 = 1:nen
    i = LM(loop1,e);
    f(i) =  f(i) + fe(loop1);   % assemble nodal force vector
    for loop2 = 1:nen
        j = LM(loop2,e);
        K(i,j) = K(i,j) + ke(loop1,loop2);  % assemble stiffness matrix
    end
end
```

naturalBC.m

```
% compute and assemble nodal boundary force vector
function f = naturalBC(f);
include_flags;

for i = 1:nnp
    if flags(i) == 1
        node     = ID(i);
        f(node) = f(node) + CArea(node)*n_bc(node);
    end
end
```

postprocessor.m

```
% postprocessing
function postprocessor(d)
include_flags;

    fprintf(1,'\n        Print stresses at the Gauss points \n')
    fprintf(1,'Element\t\t x(gauss1) \t\t x(gauss2) \t\t stress(gauss1) \t\t stress(gauss2)\n')
    fprintf(1,'--------------------------------------------------------------------------------- \n')

    % loop over elements to compute the stresses
    for e = 1:nel
    % compute stresses and displacements for the current element
```

```
    disp_and_stress(e,d);
end
```

12.5 MATLAB finite element program for heat conduction in 2D

In Section 12.2 the basic finite element program structure was introduced for one- and two- dimensional analysis of truss structures. In Section 12.3 a more general finite element program structure for one-dimensional problems was developed. In this section we describe a finite element program for scalar field problems in two-dimensions focusing on heat conduction. You will notice that the program structure is very similar to that introduced for one-dimensional problems. A brief description of various functions is provided below.

Main program: *heat2D.m*
The main program is given in *heat2D.m* file. The finite element program structure consists of the following steps:

- preprocessing
- evaluation of element conductance matrices, element nodal source vectors and their assembly
- adding the contribution from point sources and nodal boundary flux vector
- solution of the algebraic system of equations
- postprocessing

heat2d.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Heat conduction in 2D (Chapter 8)       %
% Haim Waisman, Rensselaer                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %
clear all;
close all;

% Include global variables
include_flags;

% Preprocessing
[K,f,d] = preprocessor;

% Evaluate element conductance matrix, nodal source vector and assemble
for e = 1:nel
    [ke, fe] = heat2Delem(e);
    [K,f] = assembly(K,f,e,ke,fe);
end

%  Compute and assemble nodal boundary flux vector and point sources
f = src_and_flux(f);

% Solution
 [d,f_E] = solvedr(K,f,d);

% Postprocessing
```

```
postprocessor(d);
```

Preprocessing: *preprocessor.m*

In the preprocessing phase, the input file (***input_file***), which defines material properties, mesh data, vector and matrices initializations, essential and natural conditions, point sources, the required output, is defined by the user. In the implementation provided here, fluxes are prescribed along element edges and are defined by nodal values interpolated using shape functions. The ***n_bc*** array is used for the fluxes data structure. For the heat conduction problem given in Example 8.1 with 16 quadrilateral elements (see Figure 8.9), the ***n_bc*** array is defined as follows:

$$n\_bc = \begin{bmatrix} 21 & 22 & 23 & 24 \\ 22 & 23 & 24 & 25 \\ 20.0 & 20.0 & 20.0 & 20.0 \\ 20.0 & 20.0 & 20.0 & 20.0 \end{bmatrix}$$

The number of columns corresponds to the number of edges (specified by *nbe)* on the natural boundary; the first and second rows indicate the first and the second node numbers that define the element edge; the third and fourth rows correspond to the respective nodal flux values. Note that a discontinuity in fluxes at the element boundaries could be prescribed. The input files for the 1-element and 64-element meshes are given on the website.

In the preprocessing phase, the finite mesh is generated and the working arrays ***IEN***, ***ID*** and ***LM*** are defined. The mesh generation function ***mesh2d*** utilizes MATLAB's built-in function ***linspace*** (see Chapter 1) for bisection of lines.

preprocessor.m

```
function  [K,f,d] = preprocessor;
include_flags;

% read input file
%input_file_1ele;
input_file_16ele;
%input_file_64ele;

% generate ID and LM arrays
d = setup_ID_LM(d);
```

input_file_16ele.m

```
% Input file for Example  8.1 (16-element mesh)
```

```
% material properties
k         = 5;                      % thermal conductivity
D         = k*eye(2);               % conductivity matrix

% mesh specifications
nsd       = 2;                      % number of space dimensions
nnp       = 25;                     % number of nodes
nel       = 16;                     % number of elements
nen       = 4;                      % number of element nodes
ndof      = 1;                      % number of degrees-of-freedom per node
neq       = nnp*ndof;               % number of equations

f         = zeros(neq,1);           % initialize nodal flux vector
d         = zeros(neq,1);           % initialize nodal temperature vector
K         = zeros(neq);             % initialize conductance matrix

flags     = zeros(neq,1);           % array to set B.C flags
e_bc      = zeros(neq,1);           % essential B.C array
n_bc      = zeros(neq,1);           % natural B.C array
P         = zeros(neq,1);           % initialize point source vector defined at a node
s         = 6*ones(nen,nel);        % heat source defined over the nodes

ngp    = 2;                         % number of Gauss points in each direction

% essential B.C.
flags(1:5)        = 2;              e_bc(1:5)        = 0.0;
flags(6:5:21)     = 2;              e_bc(6:5:21)     = 0.0;
nd        = 9;                      % number of nodes on essential boundary

 % what to plot
compute_flux     = 'yes';
plot_mesh        = 'yes';
plot_nod         = 'yes';
plot_temp        = 'yes';
plot_flux        = 'yes';

% natural B.C  - defined on edges positioned on the natural boundary
n_bc      = [ 21    22    23    24            % node1
              22    23    24    25            % node2
              20    20    20    20            % flux value at node 1
              20    20    20    20 ];         % flux value at node 2
nbe = 4;                                      % number of edges on the natural boundary

% mesh generation
 mesh2d;
```

mesh2d.m

```
function mesh2d;
include_flags;

lp        = sqrt(nnp);              % number of nodes in x and y direction
x0        = linspace(0,2,lp);       % equal bisection of the x nodes
```

```
y0        = 0.5*x0/2;                % y coordinates of the bottom edge
x         = [];
for i = 1:lp
      x  = [x x0];                   % define x coordinates
     y1 = linspace(y0(i),1,lp);      % bisection of y coordinates starting from a new location
     y(i:lp:lp*(lp-1)+i)     = y1;   % define y coordinates
end

% generate connectivity array IEN
rowcount = 0;
for elementcount = 1:nel
        IEN(1,elementcount)      = elementcount + rowcount;
        IEN(2,elementcount)      = elementcount + 1 + rowcount;
        IEN(3,elementcount)      = elementcount + (lp + 1) + rowcount;
        IEN(4,elementcount)      = elementcount + (lp) + rowcount;
        If mod(elementcount,lp-1) == 0
                rowcount         = rowcount + 1;
        end
end

% plot mesh and natural boundary
plotmesh;
```

plotmesh.m

```
function plotmesh;
include_flags;

if strcmpi(plot_mesh,'yes')==1;
% plot natural BC
for i=1:nbe

     node1 = n_bc(1,i);                  % first node
     node2 = n_bc(2,i);                  % second node
     x1 = x(node1); y1=y(node1);         % coordinates of the first node
     x2 = x(node2); y2=y(node2);         % coordinates of the second node

     plot([x1 x2],[y1 y2],'r','LineWidth',4); hold on
end

legend('natural B.C. (flux)');

for i = 1:nel
     XX = [x(IEN(1,i)) x(IEN(2,i)) x(IEN(3,i)) x(IEN(4,i)) x(IEN(1,i))];
     YY = [y(IEN(1,i)) y(IEN(2,i)) y(IEN(3,i)) y(IEN(4,i)) y(IEN(1,i))];
     plot(XX,YY);hold on;

     if strcmpi(plot_nod,'yes')==1;
        text(XX(1),YY(1),sprintf('%0.5g',IEN(1,i)));
        text(XX(2),YY(2),sprintf('%0.5g',IEN(2,i)));
        text(XX(3),YY(3),sprintf('%0.5g',IEN(3,i)));
        text(XX(4),YY(4),sprintf('%0.5g',IEN(4,i)));
     end
  end
```

```
end

fprintf(1,'  Mesh Params \n');
fprintf(1,'No. of Elements  %d \n',nel);
fprintf(1,'No. of Nodes     %d \n',nnp);
fprintf(1,'No. of Equations %d \n\n',neq);
```

include_flags.m

```
% file to include global variables
global   ndof nnp nel nen nsd neq ngp nee neq
global   nd e_bc s P D
global   LM ID IEN flags n_bc
global   x y nbe
global   compute_flux plot_mesh plot_temp plot_flux plot_nod
```

Element conductance matrix and nodal source flux vector: *heat2Delem.m*

This function is used for the integration of the quadrilateral element conductance matrix and nodal source vector using Gauss quadrature. The integration is carried over the parent element domain. The shape functions are computed in *Nmatheat2D* and their derivatives along with the Jacobian matrix and its determinant are computed in *Bmatheat2D*. The source is obtained by interpolation from nodal values.

heat2Delem.m

```
% Quadrilateral element conductance matrix and nodal source vector
function [ke, fe] = heat2Delem(e)
include_flags;

ke       = zeros(nen,nen);        % initialize element conductance matrix
fe       = zeros(nen,1);          % initialize element nodal source vector

% get coordinates of element nodes
je       = IEN(:,e);
C        = [x(je); y(je)]';

[w,gp]   = gauss(ngp);            % get Gauss points and weights

% compute element conductance matrix and nodal source vector
for i=1:ngp
  for j=1:ngp
     eta = gp(i);
     psi = gp(j);

     N          = NmatHeat2D(eta,psi);           % shape functions matrix
     [B, detJ]  = BmatHeat2D(eta,psi,C);         % derivative of the shape functions

     ke  = ke + w(i)*w(j)*B'*D*B*detJ;   % element conductance matrix
     se  = N*s(:,e);                      % compute s(x)
     fe  = fe + w(i)*w(j)*N'*se*detJ;     % element nodal source vector

  end
```

```
end
```

NmatHeat2D.m

```
% Shape function
function N = NmatHeat2D(eta,psi)

N        = 0.25 * [(1-psi)*(1-eta)  (1+psi)*(1-eta)   (1+psi)*(1+eta)  (1-psi)*(1+eta)];
```

BmatHeat2D.m

```
% B matrix function

function [B, detJ] = BmatHeat2D(eta,psi,C)

% calculate the Grad(N) matrix
GN       = 0.25 * [eta-1   1-eta    1+eta   -eta-1;
                   psi-1   -psi-1   1+psi    1-psi];

J        = GN*C;          % Get the Jacobian matrix
detJ     = det(J);        % Jacobian
B        = J\GN;           % compute the B matrix
```

Point sources and nodal boundary flux function: *src_and_flux.m*
This function adds the contribution of point sources *P* (prescribed at nodes only) and the boundary flux vector to the global flux vector. The *ID* array is used to relate the initial and reordered node numbering. To calculate the nodal boundary flux vector, the function loops over all boundary edges *nbe* and performs one-dimensional integration using Gauss quadrature. The integration is performed by transforming the boundary edge to the parent domain. The boundary flux vector is then assembled to the global nodal flux vector using the *ID* array. Note that f_G has a minus sign based on **Error! Reference source not found.**.

src_and_flux.m

```
% - Compute and assemble nodal boundary flux vector and point sources
function f = src_and_flux(f);
include_flags;

% assemble point sources to the global flux vector
f(ID)      = f(ID) + P(ID);

% compute nodal boundary flux vector
for i = 1:nbe

       fq           = [0 0]';                 % initialize the nodal source vector
       node1        = n_bc(1,i);              % first node
       node2        = n_bc(2,i);              % second node
       n_bce        = n_bc(3:4,i);            % flux values at an edge
```

```
    x1 = x(node1);      y1 = y(node1);   % coordinates of the first node
    x2 = x(node2);      y2 = y(node2);   % coordinates of the second node

    leng        = sqrt((x2-x1)^2 + (y2-y1)^2);     % length of an edge
    J           = leng/2;                           % 1D Jacobian
    [w,gp]      = gauss(ngp);             % get Gauss points and weights

    for i=1:ngp                            % integrate along the edge

       psi      = gp(i);
       N        = 0.5*[1-psi  1+psi];      % 1D shape functions in the parent domain
       flux     = N * n_bce;               % interpolate flux using shape functions
       fq       = fq + w(i)*N' *flux*J;    % nodal flux
    end
    fq  = -fq;                             % define nodal flux vectors as negative

    % assemble the nodal flux vector
    f(ID(node1)) = f(ID(node1)) + fq(1) ;
    f(ID(node2)) = f(ID(node2)) + fq(2);

end
```

The postprocessing: *postprocessor.m*
The postprocessing is the final phase of the finite element method. The results are plotted in Figures 8.10-8.12 in Chapter 8.

postprocess.m

```
% plot temperature and flux
function postprocess(d);
include_flags

% plot the temperature field
if strcmpi(plot_temp,'yes') == 1;
  d1      = d(ID);
  figure(2);
  for e = 1:nel
     XX = [x(IEN(1,e))  x(IEN(2,e))  x(IEN(3,e))  x(IEN(4,e))  x(IEN(1,e))];
     YY = [y(IEN(1,e))  y(IEN(2,e))  y(IEN(3,e))  y(IEN(4,e))  y(IEN(1,e))];
     dd  = [d1(IEN(1,e)) d1(IEN(2,e)) d1(IEN(3,e)) d1(IEN(4,e)) d1(IEN(1,e))];
     patch(XX,YY,dd);hold on;
  end
title('Temperature distribution'); xlabel('X'); ylabel('Y'); colorbar;
end

%compute flux vector at Gauss points
if strcmpi(compute_flux,'yes')==1;
   fprintf(1,'\n                    Heat Flux at Gauss Points \n')
   fprintf(1,'----------------------------------------------------------------------------- \n')
```

```
    for e  = 1:nel
    fprintf(1,'Element  %d \n',e)
    fprintf(1,'-------------\n')
       get_flux(d,e);
    end
end
```

get_flux.m

```
function get_flux(d,e);
include_flags;

de        = d(LM(:,e));    % extract temperature at element nodes

% get coordinates of element nodes
je         = IEN(:,e);
C          = [x(je); y(je)]';

[w,gp]   = gauss(ngp);   % get Gauss points and weights

% compute flux vector
ind = 1;
for i=1:ngp
  for j=1:ngp
      eta = gp(i);  psi = gp(j);

      N              = NmatHeat2D(eta,psi);
      [B, detJ]     = BmatHeat2D(eta,psi,C);

      X(ind,:)       =  N*C;           % Gauss points in physical coordinates
      q(:,ind)       =  -D*B*de;       % compute flux vector
      ind             = ind + 1;
  end
end
q_x = q(1,:);
q_y = q(2,:);

%          #x-coord     y-coord    q_x(eta,psi)  q_y(eta,psi)
flux_e1  = [X(:,1)       X(:,2)       q_x'             q_y'];
fprintf(1,'\t\t\tx-coord\t\t\t\ty-coord\t\t\t\tq_x\t\t\t\t\tq_y\n');
fprintf(1,'\t\t\t%f\t\t\t%f\t\t\t%f\t\t\t%f\n',flux_e1');

if strcmpi(plot_flux,'yes')==1 & strcmpi(plot_mesh,'yes') ==1;
    figure(1);
    quiver(X(:,1),X(:,2),q_x',q_y','k');
    plot(X(:,1),X(:,2),'rx');
    title('Heat Flux');
    xlabel('X');
    ylabel('Y');
end
```

Functions, which are identical to those in Chapter 5:

setup_ID_LM.m, assembly.m, solvedr.m

disp_and_stress.m

```
% compute stresses and displacements
function disp_and_stress(e,d)
include_flags;

de       = d(LM(:,e));           % extract element nodal displacements
IENe     = IEN(:,e);             % extract element connectivity information
xe       = x(IENe);              % extract element coordinates
J        = (xe(nen) - xe(1))/2;  % Jacobian
[w , gp] = gauss(ngp);           % Gauss points and weights

  % compute stresses at Gauss points
for i = 1:ngp
     xt  = 0.5*(xe(1)+xe(nen))+J*gp(i);  % location of Gauss point in the physical coordinates
     gauss_pt(i) = xt;                   % store Gauss point information

     N  = Nmatrix1D(xt,xe);      % extract shape functions
     B  = Bmatrix1D(xt,xe);      % extract derivative of shape functions

    Ee  = N*E(IENe);             % Young's modulus at element Gauss points
    stress_gauss(i)  = Ee*B*de;  % stresses at Gauss points
end

% print stresses at element Gauss points
fprintf(1,'%d\t\t\t%f\t\t\t%f\t\t\t%f\t\t\t%f\n',e,gauss_pt(1),gauss_pt(2),stress_gauss(1),stress_gauss
(2));

% compute displacements and stresses
xplot    = linspace(xe(1),xe(nen),nplot);     % equally distributed coordinate within an element

for i = 1:nplot
    xi    = xplot(i);            % x coordinate
    N     = Nmatrix1D(xi,xe);    % shape function value
    B     = Bmatrix1D(xi,xe);    % derivative of shape functions

    Ee    = N*E(IENe);           % Young's modulus
    displacement(i) = N*de;      % displacement output
    stress(i)      = Ee*B*de;    % stress output
  end

% plot displacements and stress
figure(2)
subplot(2,1,1);
plot(xplot,displacement); legend('sdf'); hold on;
ylabel('displacement');  title('FE analysis of 1D bar');

subplot(2,1,2); plot(xplot,stress); hold on;
ylabel('stress'); xlabel('x');
legend('FE');
```

ExactSolution.m

```
% plot the exact stress
function ExactSolution
include_flags;

% divide the problem domain into two regions
xa = 2:0.01:5;
xb = 5:0.01:6;

subplot(2,1,1);
% exact displacement for xa
c1 = 72;  c2 = 1 - (c1/16)*log(2);
u1 = -.5*xa + (c1/16)*log(xa) + c2;
% exact displacement for xb
c3 = 48;  c4 = log(5)/16*(c1-c3) + c2;
u2 = -.5*xb + (c3/16)*log(xb) + c4;
% plot displacement
h = plot([xa xb],[u1 u2], '--r' );
legend(h,'exact');

subplot(2,1,2);
% exact stresses for xa
ya = (36-4*xa)./xa;
% exact stress for xb
yb = (24-4*xb)./xb;
% plot stresses
plot([xa xb],[ya yb], '--r' );
```

Functions provided in Chapter 4: Nmatrix1D.m, Bmatrix1D.m, gauss.m
Functions provided in Chapter 2: solvedr.m

12.6 2D Elasticity FE Program with MATLAB

In this chapter, we introduce the finite element program for the two-dimensional linear elasticity problems. Only 4-node quadrilateral element is implemented. In Problems 9-6 and 9-7 in Chapter 9, students are assigned to implement the 3-node and 6-node triangular elements.

Main file: *elasticity2D.m*
The main program is given in *elasticity2D.m* file. The FE program structure consists of the following steps:

- preprocessing
- evaluation of element stiffness matrices and element body force vectors and assembly
- assembly of point forces (point forces defined at nodes only)
- evaluation and assembly of nodal boundary force vector
- solution of the algebraic system of equations
- postprocessing

elasticity2D.m

```
%%%%%%%%%%%%%%%%%%
% 2D Elasticity (Chapter 9)       %
% Haim Waisman                    %
%%%%%%%%%%%%%%%%%%
clear all;
close all;

% include global variables
include_flags;

% Preprocessing (same as in Chapter 8)
[K,f,d] = preprocessor;

% Element computations and assembly
for e = 1:nel
   [ke, fe] = elast2Delem(e);
   [K,f] = assembly(K,f,e,ke,fe);      % (same as in Chapter 8)
end

% Compute and assemble point forces and boundary force vector
f = point_and_trac(f);

% Solution (same as in Chapter 8)
 [d,r] = solvedr(K,f,d);

% Postprocessing (same as in Chapter 8)
postprocessor(d);
```

Input file: *input_file.m*

The data for natural B.C. is given in *n_bc* array. For example, for the 16-element mesh in Example 9.3 in Chapter 9, the *n_bc* array is given as

$$n\_bc = \begin{bmatrix} 21 & 22 & 23 & 24 \\ 22 & 23 & 24 & 25 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ -20.0 & -20.0 & -20.0 & -20.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ -20.0 & -20.0 & -20.0 & -20.0 \end{bmatrix}$$

The number of columns indicates the number of edges that lie on the natural boundary (specified by *nbe*). The first and second rows indicate the first and the second node that define of an element edge. The third and fourth rows correspond to the appropriate traction values in x and y directions at the first node, respectively, whereas rows fifth and sixth correspond to tractions in x and y directions specified at the second node. Input files for the 1 and 64 element meshes are given on the program website.

Input_file_16ele.m

```
% Input Data for Example 9.3 (16-element mesh)

% material properties
E          = 30e6;                                  % Young's modulus
ne         = 0.3;                                   % Poisson's ratio
D          = E/(1-ne^2) * [  1    ne      0         % Hooke's  law – Plane stress
                            ne     1      0
                             0     0    (1-ne)/2];

% mesh specifications
nsd        = 2;              % number of space dimensions
nnp        = 25;             % number of nodal nodes
nel        = 16;             % number of elements
nen        = 4;              % number of element nodes
ndof       = 2;              % degrees-of-freedom per node
neq        = nnp*ndof;       % number of equations

f          = zeros(neq,1);   % initialize nodal force vector
d          = zeros(neq,1);   % initialize nodal displacement matrix
K          = zeros(neq);     % initialize stiffness matrix

counter          = zeros(nnp,1);          % counter of nodes for the stress plots
nodestress       = zeros(nnp,3);          % nodal stress values for plotting [sxx syy sxy]

flags      = zeros(neq,1);   % an array to set B.C flags
e_bc       = zeros(neq,1);   % essential B.C array
n_bc       = zeros(neq,1);   % natural B.C array

P          = zeros(neq,1);              % point forces applied at nodes
b          = zeros(nen*ndof,nel);       % body force values defined at nodes

ngp        = 2;          % number of gauss points in each direction
nd         = 10;         % number of dofs on essential boundary (x and y)

% essential B.C.
ind1       = 1:10:(21-1)*ndof+1;      % all x dofs along the line y=0
ind2       = 2:10:(21-1)*ndof+2;      % all y dofs along the line x=0
flags(ind1)        = 2;               e_bc(ind1)        = 0.0;
flags(ind2)        = 2;               e_bc(ind2)        = 0.0;

% plots
compute_stress = 'yes';
plot_mesh          = 'yes';           % (same as in Chapter 8)
plot_nod           = 'yes';
plot_disp          = 'yes';
plot_stress        = 'yes';
plot_stress_xx     = 'yes';
plot_mises         = 'yes';
fact               = 9.221e3;         % factor for scaled displacements plot

% natural B.C  - defined on edges
n_bc    = [  21      22     23      24          % node1
             22      23     24      25          % node2
```

```
              0       0      0       0        %  traction value given at node1 in x
            -20     -20    -20     -20        %  traction value given at node1 in y
              0       0      0       0        %  traction value given at node2 in x
            -20     -20    -20     -20];      %  traction value given at node2 in y
nbe = 4;                                      %  number of edges on natural boundary

% mesh generation
 mesh2d;
```

include_flags.m

```
% file to include global variables
global   ndof nnp nel nen nsd neq ngp nee neq
global   nd e_bc P b D
global    LM ID IEN flags n_bc
global   x y nbe counter nodestress
global   compute_stress plot_mesh plot_disp plot_nod
global    plot_stress_xx plot_mises fact
```

Node Renumbering: *setup_ID_LM.m*

The generation of the *ID* array is similar to that in heat conduction problem with the exception that *nd* defines the number of degrees-of-freedom on the essential boundary. The *LM* array is a pointer to the renumbered degrees-of-freedom. For this purpose, we treat every node as a block consisting of two degrees-of-freedom. We define a pointer, denoted as *blk* to the beginning of each block and loop over all degrees-of-freedom *ndof* in that block.

setup_ID_LM.m

```
function  d=setup_ID_LM(d);
include_flags;

count = 0; count1 = 0;
for i = 1:neq
   if flags(i) == 2              % check if a node on essential boundary
      count   = count + 1;
      ID(i)   = count;            % arrange essential B.C nodes first
      d(count)= e_bc(i);         % store essential B.C in reordered form (d_bar)
   else
      count1 = count1 + 1;
      ID(i) = nd + count1;
   end
end

for i = 1:nel
   n = 1;
   for j = 1:nen
      blk     = ndof*(IEN(j,i)-1);
      for k = 1:ndof
         LM(n,i) = ID( blk + k  );   % create the LM matrix
         n = n + 1;
      end
   end
```

```
end
```

Element stiffness and forces function: *elast2Delem.m*

The *elast2Delem* function for numerical integration of the element stiffness matrix and element nodal body force vector remains the same as for heat conduction code except for the shape functions *NmatElast2D* and their derivatives *BmatElast2D*.

elast2Delem.m

```
function [ke, fe] = elast2Delem(e)
include_flags;

ke      = zeros(nen*ndof,nen*ndof);     % initialize element stiffness matrix
fe      = zeros(nen*ndof,1);            % initialize element body force vector

% get coordinates of element nodes
je      = IEN(:,e);
C       = [x(je); y(je)]';

[w,gp]  = gauss(ngp);                   % get gauss points and weights

% compute element stiffness and nodal force vector
for i=1:ngp
  for j=1:ngp
     eta = gp(i);
     psi = gp(j);
     N          = NmatElast2D(eta,psi);          % shape functions
     [B, detJ]  = BmatElast2D(eta,psi,C);        % derivative of the shape functions
     ke         = ke + w(i)*w(j)*B'*D*B*detJ;    % element stiffness matrix
     be         = N*b(:,e);                      % interpolate body forces using shape functions
     fe = fe + w(i)*w(j)*N'*be*detJ;             % element body force vector

  end
end
```

NmatElas2D.m

```
% Shape functions for 2D elasticity defined in parent element coordinate system
function N = NmatElast2D(eta,psi)

N1      = 0.25*(1-psi)*(1-eta);
N2      = 0.25*(1+psi)*(1-eta);
N3      = 0.25*(1+psi)*(1+eta);
N4      = 0.25*(1-psi)*(1+eta);

N       =     [N1   0    N2    0    N3   0     N4   0;       % shape functions
                0   N1   0    N2    0    N3    0   N4];
```

BmatElas2D.m

```
% B matrix function for 2D elasticity
function [B, detJ] = BmatElast2D(eta,psi,C)
```

```
% Calculate the Grad(N) matrix
 GN = 0.25 * [eta-1  1-eta   1+eta   -eta-1;
                    psi-1  -psi-1  1+psi    1-psi];
 J    = GN*C;        % Get the Jacobian matrix

 detJ  = det(J);     % compute Jacobian

 BB     = J\GN;      % compute derivatives of the shape function in physical coordinates
 B1x     = BB(1,1);
 B2x     = BB(1,2);
 B3x     = BB(1,3);
 B4x     = BB(1,4);
 B1y     = BB(2,1);
 B2y     = BB(2,2);
 B3y     = BB(2,3);
 B4y     = BB(2,4);

 B  = [ B1x     0      B2x     0      B3x     0      B4x     0  ;
         0      B1y     0      B2y     0      B3y     0      B4y;
        B1y     B1x    B2y     B2x    B3y     B3x    B4y     B4x];
```

Point forces and nodal boundary force vector function: *point_and_trac.m*
The function loops over *nbe* edges on the essential and performs a one-dimensional integration using Gauss quadrature. The integration is performed by transforming the boundary edge to the parent coordinate system $x \grave{I} [0,1]$. The nodal boundary force vector is then assembled to the global force vector using *ID* array. Similarly, point forces defined at the nodes are assembled into the global nodal force vector using the *ID* array.

point_and_trac.m

```
% Compute and assemble point forces and boundary force vector
function f = point_and_trac(f);
include_flags;

% Assemble point forces
f(ID)      = f(ID) + P(ID);

% Calculate the nodal boundary force vector
for i = 1:nbe

     ft           = [0 0 0 0]';              % initialize the nodal boundary vector
     node1        = n_bc(1,i);               % first node
     node2        = n_bc(2,i);               % second node
     n_bce        = n_bc(3:6,i);             % traction values

     x1 = x(node1);      y1=y(node1);    % coordinate of the first node
     x2 = x(node2);      y2=y(node2);    % coordinate of the second node
     leng         = sqrt((x2-x1)^2 + (y2-y1)^2);     % edge length
     J            = leng/2;                          % 1D Jacobian
     [w,gp]       = gauss(ngp);                      % get gauss points and weights
```

```
    for i=1:ngp                                   % perform1D numerical integration

       psi       = gp(i);
       N          = 0.5*[1-psi    0     1+psi     0;      % 1D shape functions in the parent edge
                          0     1-psi     0     1+psi];  % for interpolating tractions in x and y
       T    = N * n_bce;
       ft   = ft + w(i)*N' *T *J;                 % compute traction
    end

    % Assemble nodal boundary force vector
    ind1          = ndof*(node1-1)+1;             % dof corresponding to the first node
    ind2          = ndof*(node2-1)+1;             % dof corresponding to the second node
    f(ID(ind1))             = f(ID(ind1))    + ft(1) ;
    f(ID(ind1+1))           = f(ID(ind1+1)) + ft(2) ;
    f(ID(ind2))             = f(ID(ind2))    + ft(3) ;
    f(ID(ind2+1))           = f(ID(ind2+1)) + ft(4);

end
```

Postprocessing: *postprocessor.m*
The postprocessing function first calls *displacements* function to plot the deformed configuration based on the nodal displacements. The user sets a scaling factor in the input file to scale the deformation as shown in Figure 9.13 in Chapter 9.

To obtain the fringe or contour plots of stresses, stresses are computed at element nodes and then averaged over elements connected to the node. Alternatively, stresses can be computed at the Gauss points where they are most accurate and then interpolated to the nodes. The user is often interested not only in the individual stress components, but in some overall stress value such as Von-Mises stress. In case of plane stress, the von Mises stress is given by $\sigma_Y = \sqrt{\sigma_1^2 + \sigma_2^2 - 2\sigma_1\sigma_2}$, where σ_1 and σ_2 are principal stresses given by $\sigma_{1,2} = \frac{\sigma_x + \sigma_y}{2} \pm \sqrt{\left(\frac{\sigma_x - \sigma_y}{2}\right)^2 + \tau_{xy}^2}$. Figure 9.14 in Chapter 9 plots the s_{xx} stress contours for the 64-element mesh.

postprocessor.m

```
% deformation and stress ouput
function postprocess(d);
include_flags

% plot the deformed mesh
displacements(d);

% Compute strains and stresses at gauss points
```

```
s = zeros(neq,1);
if strcmpi(compute_stress,'yes')==1;
   fprintf(1,'\n                    Stress at Gauss Points \n')
   fprintf(1,'-------------------------------------------------------------------------- \n')
   for e=1:nel
   fprintf(1,'Element  %d \n',e)
   fprintf(1,'-------------\n')
      get_stress(d,e);
      nodal_stress(d,e);
   end
   stress_contours;
end
```

displacement.m

```
% scale and plot the deformed configuration
function displacements(d);
include_flags;


if strcmpi(plot_disp,'yes')==1;
displacement    = d(ID)*fact;      % scale displacements

% Compute deformed coordinates
 j = 1;
 for i = 1:ndof:nnp*ndof
    xnew(j) = x(j) + displacement(i);
    ynew(j) = y(j) + displacement(i+1);
    j = j + 1;
 end
% Plot deformed configuration over the initial configuration
 for e = 1:nel
   XXnew = [xnew(IEN(1,e)) xnew(IEN(2,e)) xnew(IEN(3,e)) xnew(IEN(4,e)) xnew(IEN(1,e))];
   YYnew = [ynew(IEN(1,e)) ynew(IEN(2,e)) ynew(IEN(3,e)) ynew(IEN(4,e)) ynew(IEN(1,e))];
   plot(XXnew,YYnew,'k');hold on;
 end
title('Initial and deformed structure'); xlabel('X'); ylabel('Y');
end
```

get_stress.m

```
% Compute strains and stresses at the gauss points
function get_stress(d,e);
include_flags;

de       = d(LM(:,e));    % element nodal displacements

% get coordinates of element nodes
je       = IEN(:,e);
C        = [x(je); y(je)]';
```

```
[w,gp]   = gauss(ngp);   % get gauss points and weights

% compute strains and stress at gauss points
ind = 1;
for i=1:ngp
  for j=1:ngp
     eta = gp(i);  psi = gp(j);

     N             = NmatElast2D(eta,psi);
     [B, detJ]     = BmatElast2D(eta,psi,C);

     Na            = [N(1,1) N(1,3) N(1,5) N(1,7)];
     X(ind,:)      =  Na*C;                % gauss points in the physical coordinates
     strain(:,ind) = B*de;
     stress(:,ind)= D*strain(:,ind);       % compute the stress [s_xx  s_yy  s_xy];
     ind      = ind + 1;
  end
end
e_xx = strain(1,:);         e_yy = strain(2,:);         e_xy = strain(3,:);    % strains at gauss points
s_xx = stress(1,:);         s_yy = stress(2,:);         s_xy = stress(3,:);    % stress at gauss points

%       Print           x-coord     y-coord    sigma_xx   sigma_yy    sigma_xy
stress_gauss    = [X(:,1)       X(:,2)        s_xx'        s_yy'          s_xy'  ];
fprintf(1,'\tx-coord\t\t\ty-coord\t\t\ts_xx\t\t\ts_yy\t\t\ts_xy\n');
fprintf(1,'\t%f\t\t%f\t\t%f\t\t%f\t\t%f\n',stress_gauss');
```

nodal_stress.m

```
% compute the average nodal stress values
function nodal_stress(d,e);
include_flags;

de       = d(LM(:,e));   % displacement at the current element nodes

% get coordinates of element nodes
je       = IEN(:,e);
C        = [x(je); y(je)]';

psi_val = [-1 1 1 -1];  % psi values at the nodes
eta_val = [-1 -1 1 1];  % eta values at the nodes

% Compute strains and stress at the element nodes
ind = 1;
for i=1:nen
     eta = eta_val(i);
     psi = psi_val(i);

     [B, detJ]     = BmatElast2D(eta,psi,C);
     strain(:,ind) = B*de;
     stress(:,ind)= D*strain(:,ind);       % compute the stress [s_xx  s_yy  s_xy];
     ind           = ind + 1;
end
e_xx = strain(1,:);         e_yy = strain(2,:);         e_xy = strain(3,:);    % strains at gauss points
s_xx = stress(1,:);         s_yy = stress(2,:);         s_xy = stress(3,:);    % stress at gauss points
```

```
counter(je)    = counter(je) + ones(nen,1);    % count tnumber of elements connected to the node
nodestress(je,:) = [s_xx'    s_yy'    s_xy' ];    % accumulate stresses at the node
```

Stress_contours.m

```
function stress_contours;
include_flags;

if strcmpi(plot_stress_xx,'yes')==1;
  figure(2);
  for e=1:nel
     XX = [x(IEN(1,e))  x(IEN(2,e))  x(IEN(3,e))  x(IEN(4,e))  x(IEN(1,e))];
     YY = [y(IEN(1,e))  y(IEN(2,e))  y(IEN(3,e))  y(IEN(4,e))  y(IEN(1,e))];

     sxx = nodestress(IEN(:,e),1)./counter(IEN(:,e));
     dd  = [sxx'   sxx(1)];
     patch(XX,YY,dd);hold on;
  end
  title('\sigma_x_x contours'); xlabel('X'); ylabel('Y'); colorbar
end

if strcmpi(plot_mises,'yes')==1;
  for e=1:nel
     XX = [x(IEN(1,e))  x(IEN(2,e))  x(IEN(3,e))  x(IEN(4,e))  x(IEN(1,e))];
     YY = [y(IEN(1,e))  y(IEN(2,e))  y(IEN(3,e))  y(IEN(4,e))  y(IEN(1,e))];

     sxx = nodestress(IEN(:,e),1)./counter(IEN(:,e));
     syy = nodestress(IEN(:,e),2)./counter(IEN(:,e));
     sxy = nodestress(IEN(:,e),3)./counter(IEN(:,e));

     S1 = 0.5*(sxx+syy) + sqrt( (0.5*(sxx-syy)).^2 + sxy.^2);        % first  principal stress
     S2 = 0.5*(sxx+syy) - sqrt( (0.5*(sxx-syy)).^2 + sxy.^2);        % second  principal stress
     mises = sqrt( S1.^2 + S2.^2 - S1.*S2 );                          % plane-stress case

     dd = [mises' mises(1)];

     figure(3);
     patch(XX,YY,dd);hold on;

  end
  title('Von Mises \sigma contours'); xlabel('X'); ylabel('Y'); colorbar
end
```

Functions, which are identical to those in Chapter 8:
preprocessor.m,   mesh2d.m,   plotmesh.m,   assembly.m,   solvedr.m

12.6 MATLAB finite element program for beams in 2D

In this section we describe a finite element program for beams in two-dimensions. The program structure is very similar to that introduced for one-dimensional problems in Section 12.3 with a main differences being two degrees-of freedom per node. A brief description of various functions is provided below.

beam.m

The main program is given in *beam.m* file. It can be seen that it is almost identical to that in Section 12.3.

```
%%%%%%%%%%%%%%%%%%%%%%%%
% Beam (Chapter 10)                    %
% Suleiman M. BaniHani, Rensselaer      %
% Rensselaer Polytechnic Institute       %
%%%%%%%%%%%%%%%%%%%%%%%%

clear all;
close all;

% include global variables
include_flags;

% Preprocessing
 [K,f,d]     = preprocessor;

% Element matrix computations and assembly
for e = 1:nel
    [ke,fe] = beamelem(e);
    [K, f]  = assembly(K,f,e,ke,fe);
end
% Add nodal boundary force vector
f = NaturalBC(f);

% Partition and solution
 [d,f_E] = solvedr(K,f,d);

% Postprocessing
postprocessor(d)
```

include_flags.m

```
% Include global variables
global nsd ndof nnp nel nen neq nd ngp
global CArea E leng phi xp P
global plot_beam plot_nod plot_stress
global LM IEN x y stress body
global flags ID xplot n_bc e_bc np nplot neqe
```

preprocessor.m

The preprocessor function reads input file and generates ID and LM arrays. The structure of ID array is identical to that for the scalar field problems (see for instance program in Chapter 5); The LM relates elements (columns) to equation numbers after renumbering. The LM array for Example Problem 10.1 is $LM = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \\ 4 & 6 \end{bmatrix}$.

```
% reads input data and sets up mesh information
function  [K,f,d] = preprocessor;
include_flags;

% input file to include all variables
  input_file_example10_1;

% Generate LM array
count = 0; count1 = 0;
for i = 1:neq
      if flags(i) == 2               % check if essential boundary
        count = count + 1;
        ID(i) = count;               % number first the degrees-of-freedom on essential boundary
        d(count)= e_bc(i);            % store the reordered values of essential B.C
    else
         count1 = count1 + 1;
        ID(i) = nd + count1;
    end
end
for e = 1:nel
    for j = 1:nen
        for m = 1:ndof
            ind = (j-1)*ndof + m;
            LM(ind,e) = ID(ndof*IEN(j,e) - ndof + m) ;   % create the LM matrix
        end
    end
end
```

input_file_example10_1.m

The cross-sectional area is prescribed at the nodes and interpolated using linear shape functions. The Young's modulus and body forces are assumed to be constant within one element; they are prescribed for each element. Essential and natural boundary conditions are prescribed for each degree of freedom on essential and natural boundary, respectively.

```
% Input Data for Example 10.1
nsd       = 2;                  % Number of spatial dimensions
ndof      =2;                  % Number of degrees-of-freedom per node
nnp       = 3;                  % Total number of global nodes
nel       = 2;                  % Total number of elements
nen       = 2;                  % Number of nodes in each element
neq       = ndof*nnp;           % Number of equations
neqe     = ndof*nen;           % Number of equations for each element

f           = zeros(neq,1);     % Initialize force vector
d           = zeros(neq,1);     % Initialize displacement vector
K           = zeros(neq);       % Initialize stiffness matrix

flags = zeros(neq,1);     % initialize flag vector
e_bc = zeros(neq,1);     % initialize vector of essential boundary condition
n_bc = zeros(neq,1);     % initialize vector of natural boundary condition

% Element properties
CArea    = [1     1 1]';    % Elements cross-sectional area
leng      = [8     4 ];      % Elements length
body      =  [-1     0 ]';    % body forces
E          = [1e4   1e4]';  % Young's Modulus

% gauss integration
ngp    = 2;                    % number of gauss points

% essential boundary conditions
% odd numbers for displacements; even for numbers for rotations
```

```
flags(1) = 2;   % flags to mark degrees-of-freedom located on the essential boundary
flags(2) = 2;   % flags to mark degrees-of-freedom located on the essential boundary
e_bc(1) = 0;   % value of prescribed displacement
e_bc(2) = 0;   % value of prescribed rotation
nd = 2;          % number of degrees-of-freedom on the essential boundary

% natural boundary conditions
% odd numbers for shear forces; even numbers for moments
flags(5) = 1;    % flags to mark degrees-of-freedom located on the natural boundary
n_bc(5) = -20; % value of force
flags(6) = 1;    % flags to mark degrees-of-freedom located on the natural boundary
n_bc(6) = 20;  % value of moment

% Applied point forces
P      = [-10 5]';       % array of point forces
xp     = [4 8]'  ;       % array of coordinates where point forces are applied
np     =  2      ;       % number of point forces

% output controls
plot_beam  = 'yes';
plot_nod    = 'yes';

% mesh generation
beam_mesh_10_1;
% number of points for plot
nplot=300;
```

beam_mesh_10_1.m

```
function  beam_mesh_10_1
include_flags;

% Node:  1    2    3   (origin placed at node 2)
x   =  [0.0  8.0  12.0  ];     % X coordinate
y   =  [0.0  0.0  0.0   ];     % Y coordinate

% connectivity array
IEN =  [1    2
        2    3];

% plot beam
plotbeam;
```

beamelem.m

```
% generate element stiffness matrix and element nodal body force vector
function [ke, fe] = beamelem(e)
include_flags;

IENe      = IEN(:,e);               % extract local connectivity information
xe        = x(IENe);                % extract x coordinates
J         = (xe(nen) - xe(1))/2;    % compute Jacobian
[w , gp]  = gauss(ngp);             % extract Gauss points and weights

ke = zeros(neqe,neqe);              % initialize element stiffness matrix
fe = zeros(neqe,1);                 % initialize element nodal force vector

for i = 1:ngp
   N    = NmatrixBeam(gp(i),xe);            % shape functions matrix
   B    = BmatrixBeam(gp(i),xe) *1/J^2;    % derivative of shape functions
   Ae   = [N(1) N(3)]*CArea(IENe);          % calculate cross-sectional area at element gauss points
   Ee   = E(e);                             % extract Young's modulus
```

```
    be   = body(e);                          % extract body forces
    ke = ke + w(i)*(B'*Ae*Ee*B);            % calculate element stiffness matrix
    fe = fe + w(i)*N'*be;                   % calculate element nodal force vector
end
ke = J*ke;
fe = J*fe;

% check for point forces in this element
for i=1:np              % loop over all point forces
    Pi  = P(i);          % extract point force
    xpi = xp(i);         % extract the location of point force within an element
     if xe(1)<=xpi &  xpi<xe(nen)
         fe = fe + Pi*[NmatrixBeam( ( (2*xpi-xe(1)-xe(nen))/(xe(nen) - xe(1)) ) ,xe)]';
     end
end
```

NmatrixBeam.m

```
% Shape functions in the natural coordinate s
 function N = NmatrixBeam(s,xe)
     L=xe(2)-xe(1);
     N(1)=1/4*(1-s)^2*(2+s);
     N(2)=L/8*(1-s)^2*(1+s);
     N(3)=1/4*(1+s)^2*(2-s);
     N(4)=L/8*(1+s)^2*(s-1);
```

BmatrixBeam.m

```
% Derivative of the shape functions in the natural coordinate s
function B = BmatrixBeam(s,xe)
     L=xe(2)-xe(1);
     B(1)=3/2*s;
     B(2)=L*(3/4*s-1/4);
     B(3)=-3/2*s;
     B(4)= L*(3/4*s+1/4);
```

SmatrixBeam.m

```
% Second derivative of the shape functions
function S = SmatrixBeam(s,xe)
     L=xe(2)-xe(1);
     S(1)=3/2;
     S(2)=3/4*L;
     S(3)=-3/2;
     S(4)= 3/4*L;
```

naturalBC.m

```
% compute and assemble nodal boundary force vector
function f = naturalBC(f);
include_flags;
for i = 1:neq
    if flags(i) == 1
        dof = ID(i);
        f(dof) = f(dof) + n_bc(dof);
    end
end
```

postprocessor.m

```
% postprocessing function
function postprocessor(d)
```

```
include_flags;

% loop over elements to plot displacements, moments and shear forces
for e = 1:nel

de        = d(LM(:,e));              % extract element nodal displacements
IENe      = IEN(:,e);                % extract element connectivity information
xe        = x(IENe);                 % extract element coordinates
J         = (xe(nen) - xe(1))/2;     % Jacobian
[w , gp]  = gauss(ngp);              % extract Gauss points and weights

% compute displacements, moments and shear forces
xplot     = linspace(xe(1),xe(nen),nplot);     % equally distributed coordinate within an element
xplotgauss= (2*xplot-xe(1)-xe(nen))/(xe(nen) - xe(1));

  for i = 1:nplot
      xi    = xplotgauss(i);                  % current coordinate
      N     = NmatrixBeam(xi,xe);             % shape functions
      B     = BmatrixBeam(xi,xe)*1/J^2;       % first derivative of shape functions
      S     = SmatrixBeam(xi,xe)*1/J^3;       % second derivative of shape functions
      Ee    = E(e);                           % Young's modulus
      displacement(i)   = N*de ;              % displacement output
      moment(i)         = Ee*B*de;            % moment output
      shear(i)          = Ee*S*de;            % Shear force output
  end

% plot displacements, moment and shear forces
[x_plot,S_ex,M_ex,w_ex]=exact; % call exact beam solution

figure(2)
plot(xplot,displacement,'-.r'); hold on;
plot(x_plot,w_ex,'-k'); legend('FE','Exact Solution'); hold on;
ylabel('displacement');  title('Displacements: FE versus analytical beam solutions');

figure(3)
plot(xplot,moment,'-.r'); hold on;
plot(x_plot,M_ex,'-k'); legend('FE','Exact Solution'); hold on;
ylabel('moment'); xlabel('x'); title('Moments: FE versus analytical beam solutions');

figure(4)
plot(xplot,shear,'-.r'); hold on;
plot(x_plot,S_ex,'-k'); legend('FE','Exact Solution'); hold on;
ylabel('shear'); xlabel('x');   title('Shear: FE versus analytical beam solutions');

end
```

Functions which are identical to those in Chapter 5 are: assembly.m, solvedr.m, gauss.m

Problems –Linear Algebra

Problem 1-1

Write a MATLAB program to generate a set of linear algebraic equations **Ax**=**b** where $\mathbf{A}$ is an $n \times n$ matrix given by

$$A_{ij} = \begin{cases} -1 \ \ if \ \ i = j-1 \ or \ i = j+1 \\ 2 \ if \ \ i = j \\ 0 \ otherwise \end{cases}$$

and compute $\mathbf{A}^{-1}$. Then check how closely $\mathbf{B} = \mathbf{A}^{-1}\mathbf{A}$ corresponds to $\mathbf{I}$. Do this for $n = 5,10,1000$ and a larger value you choose. The accuracy of the computed product can be compared to the correct results by computing a norm of the error given by

$$err = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}\left(B_{ij} - I_{ij}\right)^2$$

Repeat the above with the matrix $\mathbf{A}$ defined by

$$A_{ij} = \frac{1}{i+j}$$

Repeat the above with $n = 3,4,5,6,7\ldots$. Stop when the error is greater than one, since the solution is then meaningless.

The top equation is of the form we will see in finite element equations. They can be accurately solved even for very large system for they are not susceptible to roundoff error. They are known as well-conditioned. The second matrix is called a Hilbert matrix. It is an example of an estremely ill-conditioned matrix.

Problem 1-2

Consider a system of linear equations

$$\begin{cases} 8x_1 + x_2 + 6x_3 = 7.5 \\ 3x_1 + 5x_2 + 7x_3 = 4 \\ 4x_1 + 9x_2 + 2x_3 = 12 \end{cases}$$

a) Write the system of equations in matrix notation $\mathbf{Ax} = \mathbf{b}$ and solve for the unknown $\mathbf{x}$ using MATLAB
b) Suppose we impose an additional constraint on the solution $g(\mathbf{x}) = x_1 + x_2 + x - 1 = 0$. Using MATLAB find a new vector $\mathbf{x}^{new}$ so that it will satisfy exactly the constraint equation $g(\mathbf{x}^{new})$ and will minimize the error $err(\mathbf{x}^{new}) = (\mathbf{Ax}^{new} - \mathbf{b})^T(\mathbf{Ax}^{new} - \mathbf{b})$

Problem 1-3: Consider the following symmetrix matrices $\mathbf{K}$:

$$\begin{bmatrix} k_1 + k_2 & -k_2 \\ -k_2 & k_2 \end{bmatrix} \qquad \begin{bmatrix} k & 0 \\ 0 & k \end{bmatrix} \qquad \begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & k_1 + k_2 & -k_2 \\ 0 & -k_2 & k_2 \end{bmatrix}$$

where k, k_1, k_2 are positive constants.

a) Check if the above three matrices are positive definite. Recall that if for any vector $\mathbf{x} \neq 0$ we have $\mathbf{x}^T\mathbf{Kx} > 0$ then matrix $\mathbf{K}$ is called Symmetric Positive Definite (SPD). If, on the other hand, $\mathbf{x}^T\mathbf{Kx} \geq 0$ for any vector $\mathbf{x} \neq 0$ then the matrix $\mathbf{K}$ is symmetric semi-positive definite. Choose one of the semi-positive definite matrices shown above and show that for any right hand side vector, $\mathbf{f}$, the system of equations $\mathbf{Kd} = \mathbf{f}$ has no unique solution.
b) Verify your results by computing the eigenvalues for the above three matrices