Our aim in the ME623A workshops is to develop a general purpose framework for a Finite Element code in Matlab. The code needs to be structured in a manner so as to take optimum advantage of the generality of the FE structure. This requires a bit of programming discipline.

The following functions, to be coded in Matlab, are designed to guide you to a working code, that, though not completely general, is adequate for solving a large variety of linear elastic problems in 1, 2, or 3 dimensions. All variables described herein are in *italics* and functions are in `typewriter` fonts. Maintain the size of all variables exactly as directed below. The code will be developed over many sessions and a mistake now will lead to trouble later.

We will suggest verification problems that will be useful for checking if a function that you have written actually works properly. Remember a few thumb rules:

1. Never trust your own code, always try to devise problems that can be used to crash test it. You should always try to make your own function crash under some input. Devising problems to test a function that you have written, so as to check when it fails, is one of the most challenging jobs in FE coding.

2. Follow the array dimensions carefully.

3. Insert checks on values based on physical considerations, eg. number of space dimensions cannot be more than 3. An error should be thrown up by the code if the user tries to enter 4.

Here are the steps that we will have to accomplish.

Step 1: The basic variables in a linear elastic FE code are

(a) Number of nodes in the system: *nnode*

(b) Number of elements in the system: *nelem*

(c) Number of different materials in the system: *nmat*

(d) Number of degrees of freedom per node: *ndof*

(e) Number of nodes per element: *nperlm*

(f) Number of space dimensions: *ndim*

Step 2: Input data for an entire analysis is generally read in from a text file. All input data resides in a single text file. This includes nodal coordinates, element connectivity and material property information, input material properties, boundary conditions (free or fixed), boundary condition values, etc. It is a good idea to design your input file such that it has separate sections for each type of data. The task in this step is to

(a) Design a input file. Decide what goes where, eg. you may first have all nodal coordinates, followed by element connectivity, followed by material properties, nodal forces and then boundary conditions. Remember that you may have to read in more things later. Also remember,

i. node numbers will go from $1 : nnode$ and element numbers from $1 : nelem$.

ii. there will be *ndim* values of coordinates for each node

iii. there will be *nperlm* values for nodes connected to each element. Also, you might want to read in the material number which the element is made up of, which goes from $1 : nmat$.

iv. boundary conditions can be specified (as zero or non-zero) for all $1 : nnode$ nodes for $1 : ndof$ degrees of freedom. However, only a few nodes will have boundary conditions. The same holds for external forces.

v. We will have to input material properties of all $1 : nmat+1$ materials in the system. The first entry for each material is the type *mat_type_number*, followed by the values of the requisite properties. Each of these material types will require some number of material constants to be specified. For example, if you have a linear elastic isotropic material you will have to specify E, ν for it, while if an element is a spring element, you specify only one material property, its stiffness. Each material type (eg. for linear elastic, truss, beam bending etc) will be given an unique *mat_type_number*. We will assume that it goes from $1 : 7$ i.e. we can have a maximum of seven different material types.

vi. Thus, you will have to design what is known as a 'material library'. It is a function that is of the form

$$[num\_mat\_data] = \texttt{mat\_lib}(mat\_type\_number)$$

Here num_mat_data is the number of data entries that should be read in for a given mat_type_number. For instance, if $mat\_type\_number = 1$ denotes a linear elastic isotropic material, then $num\_mat\_data = 2$. For transversly isotropic material, it is 5.

(b) Write a function that has the following structure:

$$[x, conn, bound, force, mate] = \texttt{read\_input}(nnode, nelem, nmat, ndof, nperlm, ndim)$$

Here the output variables are:

i. $x(1 : nnode, 1 : ndim)$: array containing nodal coordinates

ii. $conn(1 : nelem, 1 : (nperlm + 1))$: array containing material number associated with an element and nodes connected to it.

iii. $bound(1 : nnode, 1 : ndof)$: array containing boundary conditions, a dof is assigned 0 if it is free and 1 if it is fixed.

iv. $force(1 : nnode, 1 : ndof)$: if a degree of freedom has specified displacement, it will contain the value of the displacement. If it has a specified force, it will contain the value of the force at that dof. Recall that the same dif cannot have both force and displacemet specified.

v. $mate(1 : nelem, 1 : nmat)$: material number assigned to an element.

vi. $matdata(1 : nmat, 1 : \texttt{mat\_lib}(mat\_type\_number))$: property data for the material type.

(c) Check. One way could be to write an input file for the truss problem done in the class. Is your code able to read all the information in? Is the information being stored in the arrays as you wanted them to be?

(d) Another check. Suppose you have a problem in 3 dimensions with one 8 noded element with 3 dofs per node. The element is made up of an elastic material. The element is shaped like a cube of sides 1 units with nodes at the vertices. The origin of the global axes is located at one of the lower nodes. At each of the top 4 nodes, a force of 0.25 units is applied while the faces $x = y = z = 0$ are not allowed to move in directions normal to them. The material of the cube is linear elastic, with $E = 100$ units and $\nu = 0.3$. Write an input file for this problem and check if your `read_input` function can read this correctly.

Try to decide the sizes of the *x,conn,bound, force, mate, matdata* arrays and check if they all look as they should.

Step 3: Generate the destination array for the mesh that you have read in. The destination *destination* array is of the size $destination(1 : nelem, 1 : nperlm * ndof)$. Your function should be

$$[destination] = \texttt{form\_destination}(conn, nelem, nnode, nperlm, ndof)$$

The destination array stores, for each element, for each local dof in the element, the number of the global dof to which the local dof corresponds.

Step 4: Initialise a few arrays: $displacements(1 : nnode * ndof), rhs(1 : nnode : ndof), global\_K(1 : nnode * ndof, 1 : nnode * ndof), global\_force(1 : nnode : ndof)$ to zeros.

Step 5: Write a function to localise variables for an element. This means that the function will, for an element e be able to extract from the global arrays the quantities corresponding to this element. It should have the following form

$$[local\_disp, local\_x, local\_force, local\_props, local\_destination] =$$
$$\texttt{localise}(e, destination, conn, x, displacements, rhs, mate, matdata, nnode,$$
$$nelem, ndim, nperlm, ndof)$$

Here

(a) $local\_disp(1 : nperlm * ndof)$: contains the displacements of the dofs attached to element e

(b) $local\_x(1 : nperlm * ndim)$: contains the nodal coordinates of the nodes attached to the element e

(c) $local\_force(1 : nperlm * ndof)$: contains the forces applied at the dofs attached to e

(d) $local\_props(1 : \texttt{mat\_lib}(mat\_type\_number))$ contains the material property values pertaining to e including the material number.

(e) $local\_destination(1 : nperlm * ndof)$ contains the destination addresses of all dofs in element e

Step 7: Now we will create the element stiffness and the rhs. This function will be different for different types of elements. So, we will start with a 'truss' element. The form of this function will be:

$$[local\_stiff, local\_rhs] =$$
$$\texttt{truss\_element}(local\_x, local\_disp, local\_force, local\_props, ndof, ndim, nperlm)$$

Here

(a) $local\_stiff(1 : nperlm * ndof, 1 : nperlm * ndof)$ is the local element stiffness matrix

(b) $local\_rhs(1 : nperlm * ndof)$ is the local rhs vector for the element type. It is a null vector for the truss without body forces.

Recall that a 2-noded truss element has $nperlm = 2, ndof = 2, ndim = 2$ and is hence 4×4. It is of the form

$$\boldsymbol{K}_{local} = \begin{pmatrix} k & 0 & -k & 0 \\ 0 & 0 & 0 & 0 \\ -k & 0 & k & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

where, k is the stiffness for the element e, to be obtained from its $local\_prop$ array. In this case, your `matdata` function should tell you that you need only one material property to be read in for a truss element, i.e. AE. You calculate the length L from the nodal coordinates in $local\_x$, so that $k = AE/L$

After, setting up $\boldsymbol{K}_{local}$, set up the transformation matrix $\boldsymbol{T}(1 : nperlm * ndof, 1 : nperlm * ndof)$. This too can be calculated (see class notes) from the array $local\_x$. The the transformation $\boldsymbol{K} = \boldsymbol{T}^T \boldsymbol{K}_{local} \boldsymbol{T}$ gives the local stiffness in the global coordinate system.

Step 6: We are now in a position to set up the main code. Let us call it `myFEMcode`. It should have the form:

$$[disp] = \texttt{myFEMcode}(nnode, nelem, nmat, ndof, nperlm, ndim)$$

It should basically consist of a series of function calls:

! read input file

$[x, conn, bound, force, mate] = \texttt{read\_input}(nnode, nelem, nmat, ndof, nperlm, ndim);$

! form destination array

$[destination] = \texttt{form\_destination}(conn, nelem, nnode, nperlm, ndof);$

Initialise arrays as instructed in Step 4

! loop over elements

for $e = 1 : nelem$

$[local\_disp, local\_x, local\_force, local\_props, , local\_destination] =$

$\texttt{localise}(e, destination, conn, x, displacements, rhs, mate, matdata, nnode,$

$nelem, ndim, nperlm, ndof)$

$[local\_stiff, local\_rhs] =$

if local_props(1) = the number designated for truss elements

$\texttt{truss\_element}(local\_x, local\_disp, local\_force, local\_props, ndof, ndim, nperlm)$

endif

$[global\_K, global\_force] = \texttt{assemble}(global\_K, global\_force,$

$local\_stiff, local\_rhs, local\_destination, switch)$

end loop over elements

Step 8: We need to write a function for assembling the local stiffness into global and $local\_rhs$ into the $global\_rhs$. This function should have the form:

$$\begin{aligned}[global\_K, global\_force] &= \texttt{assemble}(global\_K, global\_force, \\ &\quad local\_stiff, local\_rhs, local\_destination, switch)\end{aligned}$$

The assembly process is controlled by *switch* which tells the function if only global stiffness, only global rhs or both are to be assembled. The assembly process for is:

$$\begin{aligned}global\_K(local\_destination, local\_destination) &= global\_K(local\_destination, local\_destination) \\ &\quad + local\_stiff; \\ rhs(local\_destination) &= rhs(local\_destination) + local\_rhs;\end{aligned}$$

Step 9: The last function we will need to write is one for applying displacement boundary conditions on the assembled global stiffness matrix. This is going to be an involved task. Recall that we have stored two arrays $bound(1:nnode, 1:ndof)$ and $force(1:nnode, 1:ndof)$. We noted that $bound(i,j) = 1$ when dof j of node i has a specified displacement. The corresponding value of the displacement is stored in $force(i,j)$. Now we will write a function of the form:

$$[global\_K, global\_force] = \texttt{apply\_bc}(global\_K, global\_force, bound, force, nnode, ndof)$$

Assembling the $global\_force$ vector is easy.

for $i = 1 : nnode$
for $j = 1 : ndof$
if $(bound(i,j) == 0)$ then
$I = (i-1) * ndof + j$
$global\_force(I) = force(i,j)$
end for

Now to apply displacement boundary conditions, changes to the global stiffness and force are required:

for $i = 1 : nnode$
for $j = 1 : ndof$
if $(bound(i,j) == 1)$ then
$spec\_disp = force(i,j)$
$I = (i-1) * ndof + j$ (1)
$global\_force = global\_force - spec\_disp * global\_K(:, I)$
$global\_K(I, :) = 0$
$global\_K(:, I) = 0$
$global\_K(I, I) = 1$
$global\_force(I) = spec\_disp$
end for

Step 10: Now, we can solve for the displacements by

$$displacements = \texttt{inv}(global\_stiff) * global\_force$$

Step 11: Let us now play around with the code. We will now modify it to add another element type — a 2-d beam that can also deform axially. Thus, for this element, $ndof = 3, nperlm = 2$. Let the material type number mat_type_number for this element be 2. The number of properties required for it will also be 2, i.e $AE$and EI. Alter the `mat_lib` function to accomodate this, i.e $\texttt{mat\_lib}(2) = 2$.

Step 12: Now, go to the function `myFEMcode`. Add,

> if local_props(1) = the number designated for beam elements
> `beam_element`$(local\_x, local\_disp, local\_force, local\_props, ndof, ndim, nperlm)$
> endif

after the if loop on `truss_element`.

Step 13: Form a new element routine called
`beam_element`$(local\_x, local\_disp, local\_force, local\_props, ndof, ndim, nperlm)$ Here, you will form the local stiffness matrix, which will now be 6×6. The transformation matrix based on the orientation of the element will have to be appropriately defined. Note that we have done this in the class.

Step 14: We are now in a position to solve a few beam problems. We will first set up a 2 element cantilever beam with a load P at the end $x = L$ to verify that our code gives $\delta = PL^3/3EI$ at the end $x = L$.

Step 15: Note that the cantilever problem does not test the code for the axial response. Nor does it test if we have coded the stiffness transformation properly. We can check that by modelling a portal frame structure with a sideways and a downward load at the top left node as discussed in the class. Write an input file and solve this problem.

Step 16: Our next aim is to write an element routine for a 3 noded triangular element for solving 2-d plane stress or strain problems in elasticity. The element has 2 dof per node (two displacements) and 3 nodes per element. It requires two properties to be specified for isotropic linear elasticity i.e. E and ν. Use expressions for shape functions, $\boldsymbol{N}$, strain displacement $\boldsymbol{B}$ developed in the class. The area of the element needs to be calculated from the nodal coordinates.