Programing the Finite Element Method with Matlab

Jack Chesso[*]

3rd October 2002

1 Introduction

The goal of this document is to give a very brief overview and direction in the writing of finite element code using Matlab. It is assumed that the reader has a basic familiarity with the theory of the finite element method, and our attention will be mostly on the implementation. An example finite element code for analyzing static linear elastic problems written in Matlab is presented to illustrate how to program the finite element method. The example program and supporting files are available at
`http://www.tam.northwestern.edu/jfc795/Matlab/`

1.1 Notation

For clarity we adopt the following notation in this paper; the bold italics font $\boldsymbol{v}$ denotes a vector quantity of dimension equal to the spacial dimension of the problem *i.e.* the displacement or velocity at a point, the bold non-italicized font $\mathbf{d}$ denotes a vector or matrix which is of dimension of the number of unknowns in the discrete system *i.e.* a system matrix like the stiffness matrix, an uppercase subscript denotes a node number whereas a lowercase subscript in general denotes a vector component along a Cartesian unit vector. So, if $\mathbf{d}$ is the system vector of nodal unknowns, $\boldsymbol{u}_I$ is a displacement vector of node I and u_{Ii} is the component of the displacement at node I in the i direction, or $\boldsymbol{u}_I \cdot \boldsymbol{e}_i$. Often Matlab syntax will be intermixed with mathematical notation

[*]Graduate Research Assistant, Northwestern University (j-chessa@northwestern.edu)

which hopefully adds clarity to the explanation. The typewriter font, `font`, is used to indicate that Matlab syntax is being employed.

2 A Few Words on Writing Matlab Programs

The Matlab programming language is useful in illustrating how to program the finite element method due to the fact it allows one to very quickly code numerical methods and has a vast predefined mathematical library. This is also due to the fact that matrix (sparse and dense), vector and many linear algebra tools are already defined and the developer can focus entirely on the implementation of the algorithm not defining these data structures. The extensive mathematics and graphics functions further free the developer from the drudgery of developing these functions themselves or finding equivalent pre-existing libraries. A simple two dimensional finite element program in Matlab need only be a few hundred lines of code whereas in Fortran or C++ one might need a few thousand.

Although the Matlab programming language is very complete with respect to it's mathematical functions there are a few finite element specific tasks that are helpful to develop as separate functions. These have been programed and are available at the previously mentioned web site.

As usual there is a trade off to this ease of development. Since Matlab is an interpretive language; each line of code is interpreted by the Matlab command line interpreter and executed sequentially at run time, the run times can be much greater than that of compiled programming languages like Fortran or C++. It should be noted that the built-in Matlab functions are already compiled and are extremely efficient and should be used as much as possible. Keeping this slow down due to the interpretive nature of Matlab in mind, one programming construct that should be avoided at all costs is the for loop, especially nested for loops since these can make a Matlab programs run time orders of magnitude longer than may be needed. Often for loops can be eliminated using Matlab's vectorized addressing. For example, the following Matlab code which sets the row and column of a matrix `A` to zero and puts one on the diagonal

```
for i=1:size(A,2)
  A(n,i)=0;
end
for i=1:size(A,1)
  A(i,n)=0;
end
```

```
A(n,n)=1;
```

should never be used since the following code

```
A(:,n)=0;
A(:,n)=0;
A(n,n)=0;
```

does that same in three interpreted lines as opposed to $nr+nc+1$ interpreted lines, where `A` is a $nr \times nc$ dimensional matrix. One can easily see that this can quickly add significant overhead when dealing with large systems (as is often the case with finite element codes). Sometimes for loops are unavoidable, but it is surprising how few times this is the case. It is suggested that after developing a Matlab program, one go back and see how/if they can eliminate any of the for loops. With practice this will become second nature.

3 Sections of a Typical Finite Element Program

A typical finite element program consists of the following sections

1. Preprocessing section
2. Processing section
3. Post-processing section

In the preprocessing section the data and structures that define the particular problem statement are defined. These include the finite element discretization, material properties, solution parameters *etc.*. The processing section is where the finite element objects *i.e.* stiffness matrices, force vectors *etc.* are computed, boundary conditions are enforced and the system is solved. The post-processing section is where the results from the processing section are analyzed. Here stresses may be calculated and data might be visualized. In this document we will be primarily concerned with the processing section. Many pre and post-processing operations are already programmed in Matlab and are included in the online reference; if interested one can either look directly at the Matlab script files or type `help` *'function name'* at the Matlab command line to get further information on how to use these functions.

4 Finite Element Data Structures in Matlab

Here we discuss the data structures used in the finite element method and specifically those that are implemented in the example code. These are somewhat arbitrary in that one can imagine numerous ways to store the data for a finite element program, but we attempt to use structures that are the most flexible and conducive to Matlab. The design of these data structures may be depend on the programming language used, but usually are not significantly different than those outlined here.

4.1 Nodal Coordinate Matrix

Since we are programming the finite element method it is not unexpected that we need some way of representing the element discretization of the domain. To do so we define a set of nodes and a set of elements that connect these nodes in some way. The node coordinates are stored in the nodal coordinate matrix. This is simply a matrix of the nodal coordinates (*imagine that*). The dimension of this matrix is $nn \times sdim$ where nn is the number of nodes and $sdim$ is the number of spacial dimensions of the problem. So, if we consider a nodal coordinate matrix `nodes` the y-coordinate of the n^{th} node is `nodes(n,2)`. Figure 1 shows a simple finite element discretization. For this simple mesh the nodal coordinate matrix would be as follows:

$$\texttt{nodes} = \begin{bmatrix} 0.0 & 0.0 \\ 2.0 & 0.0 \\ 0.0 & 3.0 \\ 2.0 & 3.0 \\ 0.0 & 6.0 \\ 2.0 & 6.0 \end{bmatrix}. \tag{1}$$

4.2 Element Connectivity Matrix

The element definitions are stored in the element connectivity matrix. This is a matrix of node numbers where each row of the matrix contains the connectivity of an element. So if we consider the connectivity matrix `elements` that describes a mesh of 4-node quadrilaterals the 36th element is defined by the connectivity vector `elements(36,:)` which for example may be `[ 36 42 13 14]` or that the elements connects nodes $36 \rightarrow 42 \rightarrow 13 \rightarrow 14$. So for the simple mesh in Figure 1 the element connectivity matrix is

$$\texttt{elements} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 3 \\ 4 & 5 & 2 \\ 6 & 5 & 4 \end{bmatrix}. \tag{2}$$

Note that the elements connectivities are all ordered in a counter-clockwise fashion; if this is not done so some Jacobian's will be negative and thus can cause the stiffnesses matrix to be singular (and obviously wrong!!!).

4.3 Definition of Boundaries

In the finite element method boundary conditions are used to either form force vectors (natural or Neumann boundary conditions) or to specify the value of the unknown field on a boundary (essential or Dirichlet boundary conditions). In either case a definition of the boundary is needed. The most versatile way of accomplishing this is to keep a finite element discretization of the necessary boundaries. The dimension of this mesh will be one order less that the spacial dimension of the problem (*i.e.* a 2D boundary mesh for a 3D problem, 1D boundary mesh for a 2D problem *etc.*). Once again let's consider the simple mesh in Figure 1. Suppose we wish to apply a boundary condition on the right edge of the mesh then the boundary mesh would be the defined by the following element connectivity matrix of 2-node line elements

$$\texttt{right Edge} = \begin{bmatrix} 2 & 4 \\ 4 & 6 \end{bmatrix}. \tag{3}$$

Note that the numbers in the boundary connectivity matrix refer to the same node coordinate matrix as do the numbers in the connectivity matrix of the interior elements. If we wish to apply an essential boundary conditions on this edge we need a list of the node numbers on the edge. This can be easily done in Matlab with the `unique` function.

```
nodesOnBoundary = unique(rightEdge);
```

This will set the vector `nodesOnBoundary` equal to [2 4 6]. If we wish to from a force vector from a natural boundary condition on this edge we simply loop over the elements and integrate the force on the edge just as we would integrate any finite element operators on the domain interior *i.e.* the stiffness matrix K.

4.4 Dof Mapping

Ultimately for all finite element programs we solve a linear algebraic system of the form

$$\mathbf{K}\mathbf{d} = \mathbf{f} \tag{4}$$

for the vector $\mathbf{d}$. The vector $\mathbf{d}$ contains the nodal unknowns for that define the finite element approximation

$$u^h(x) = \sum_{I=1}^{nn} N_I(x) d_I \tag{5}$$

where $N_I(x)$ are the finite element shape functions, d_I are the nodal unknowns for the node I which may be scalar or vector quantities (if $u^h(x)$ is a scalar or vector) and nn is the number of nodes in the discretization. For scalar fields the location of the nodal unknowns in $\mathbf{d}$ is most obviously as follows

$$d_I = \texttt{d(I)}, \tag{6}$$

but for vector fields the location of the nodal unknown d_{Ii}, where I refers to the node number and i refers to the component of the vector nodal unknown $\boldsymbol{d}_I$, there is some ambiguity. We need to define a mapping from the node number and vector component to the index of the nodal unknown vector $\mathbf{d}$. This mapping can be written as

$$f : \{I, i\} \rightarrow n \tag{7}$$

where f is the mapping, I is the node number, i is the component and n is the index in $\mathbf{d}$. So the location of unknown u_{Ii} in $\mathbf{d}$ is as follows

$$u_{Ii} = \mathbf{d}_{f(I,i)}. \tag{8}$$

There are two common mappings used. The first is to alternate between each spacial component in the nodal unknown vector $\mathbf{d}$. With this arrangement the nodal unknown vector $\mathbf{d}$ is of the form

$$\mathbf{d} = \begin{bmatrix} u_{1x} \\ u_{1y} \\ \vdots \\ u_{2x} \\ u_{2y} \\ \vdots \\ u_{nn\,x} \\ u_{nn\,y} \\ \vdots \end{bmatrix} \tag{9}$$

where nn is again the number of nodes in the discretization. This mapping is

$$n = sdim(I-1) + i. \tag{10}$$

With this mapping the i component of the displacement at node I is located as follows in $\mathbf{d}$

$$d_{Ii} = \texttt{d(sdim*(I-1) + i)}. \tag{11}$$

The other option is to group all the like components of the nodal unknowns in a contiguous portion of $\mathbf{d}$ or as follows

$$\mathbf{d} = \begin{bmatrix} u_{1x} \\ u_{2x} \\ \vdots \\ u_{nx} \\ u_{1y} \\ u_{2y} \\ \vdots \end{bmatrix} \tag{12}$$

The mapping in this case is

$$n = (i-1)nn + I \tag{13}$$

So for this structure the i component of the displacement at node I is located at in $\mathbf{d}$

$$d_{Ii} = \texttt{d((i-1)*nn + I)} \tag{14}$$

For reasons that will be appreciated when we discuss the *scattering* of element operators into system operators we will adopt the latter dof mapping. It is important to be comfortable with these mappings since it is an operation that is performed regularly in any finite element code. Of course which ever mapping is chosen the stiffness matrix and force vectors should have the same structure.

5 Computation of Finite Element Operators

At the heart of the finite element program is the computation of finite element operators. For example in a linear static code they would be the stiffness matrix

$$\mathbf{K} = \int_\Omega \mathbf{B}^T \, \mathbf{C} \, \mathbf{B} \, d\Omega \tag{15}$$

and the external force vector

$$\mathbf{f}^{ext} = \int_{\Gamma_t} \mathbf{N}\boldsymbol{t}\, d\Gamma. \tag{16}$$

The global operators are evaluated by looping over the elements in the discretization, integrating the operator over the element and then to *scatter* the local element operator into the global operator. This procedure is written mathematically with the Assembly operator $\mathbb{A}$

$$\mathbf{K} = \mathbb{A}_e \int_{\Omega^e} \mathbf{B}^{\mathbf{e}T}\, \mathbf{C}\, \mathbf{B}^{\mathbf{e}}\, d\Omega \tag{17}$$

5.1 Quadrature

The integration of an element operator is performed with an appropriate quadrature rule which depends on the element and the function being integrated. In general a quadrature rule is as follows

$$\int_{\xi=-1}^{\xi=1} f(\xi) d\xi = \sum_q f(\xi_q) W_q \tag{18}$$

where $f(\xi)$ is the function to be integrated, ξ_q are the quadrature points and W_q the quadrature weights. The function `quadrature` generates a vector of quadrature points and a vector of quadrature weights for a quadrature rule. The syntax of this function is as follows

```
[quadWeights,quadPoints] = quadrature(integrationOrder,
            elementType,dimensionOfQuadrature);
```

so an example quadrature loop to integrate the function $f = x^3$ on a triangular element would be as follows

```
[qPt,qWt]=quadrature(3,'TRIANGULAR',2);
for q=1:length(qWt)
  xi = qPt(q); % quadrature point
  % get the global coordinte x at the quadrature point xi
  % and the Jacobian at the quadrature point, jac
  ...
  f_int = f_int + x^3 * jac*qWt(q);
end
```

5.2 Operator "Scattering"

Once the element operator is computed it needs to be scattered into the global operator. An illustration of the scattering of an element force vector into a global force vector is shown in Figure 2. The scattering is dependent on the element connectivity and the dof mapping chosen. The following code performs the scatter indicated in Figure 2

```
elemConn = element(e,:);            % element connectivity
enn = length(elemConn);
for I=1:enn;                        % loop over element nodes
  for i=1:2                         % loop over spacial dimensions
    Ii=nn*(i-1)+sctr(I);            % dof map
    f(Ii) = f(Ii) + f((i-1)*enn+I);
  end
end
```

but uses a nested for loop (bad bad bad). This is an even more egregious act considering the fact that it occurs within an element loop so this can really slow down the execution time of the program (by orders of magnitude in many cases). And it gets even worse when scattering a matrix operator (stiffness matrix) since we will have four nested for loops. Fortunately, Matlab allows for an easy solution; the following code performs exactly the same scattering as is done in the above code but with out any for loops, so the execution time is much improved (not to mention that it is much more concise).

```
sctr = element(e,:);                % element connectivity
sctrVct = [ sctr sctr+nn ];         % vector scatter
f(sctrVct) = f(sctrVct) + fe;
```

To scatter an element stiffness matrix into a global stiffness matrix the following line does the trick

```
K(sctrVct,sctrVct) = K(sctrVct,sctrVct) + ke;
```

This terse array indexing of Matlab is a bit confusing at first but if one spends a bit of time getting used to it, it will become quite natural and useful.

5.3 Enforcement of Essential Boundary Conditions

The final issue before solving the linear algebraic system of finite element equations is the enforcement of the essential boundary conditions. Typically this involves modifying the system

$$\mathbf{Kd} = \mathbf{f} \tag{19}$$

so that the essential boundary condition

$$d_n = \bar{d}_n \tag{20}$$

is satisfied while retaining the original finite element equations on the unconstrained dofs. In (20) the subscript n refers to the index of the vector $\mathbf{d}$ not to a node number. An easy way to enforce (20) would be to modify n^{th} row of the $\mathbf{K}$ matrix so that

$$K_{nm} = \delta_{nm} \quad \forall m \in \{1, 2 \dots N\} \tag{21}$$

where N is the dimension of $\mathbf{K}$ and setting

$$f_n = \bar{d}_n. \tag{22}$$

This reduces the n^{th} equation of (19) to (20). Unfortunately, this destroys the symmetry of $\mathbf{K}$ which is a very important property for many efficient linear solvers. By modifying the n^{th} column of $\mathbf{K}$ as follows

$$K_{m,n} = \delta_{nm} \quad \forall m \in \{1, 2 \dots N\}. \tag{23}$$

We can make the system symmetric. Of course this will modify every equation in (19) unless we modify the force vector $\mathbf{f}$

$$f_m = K_{mn}\bar{d}_n. \tag{24}$$

If we write the modified k^{th} equation in (19)

$$\begin{aligned} K_{k1}d_1 + K_{k2}d_2 + \dots K_{k(n-1)}d_{n-1} + & \\ K_{k(n+1)}d_{n+1} + \dots + K_{kN}d_N &= f_k - K_{kn}\bar{d}_n \end{aligned} \tag{25}$$

it can be seen that we have the same linear equations as in (19), but just with the internal force from the constrained dof. This procedure in Matlab i s as follows

```
f = f - K(:,fixedDofs)*fixedDofValues;
K(:,fixedDofs) = 0;
K(fixedDofs,:) = 0;
K(fixedDofs,fixedDofs) = bcwt*speye(length(fixedDofs));
f(fixedDofs) = bcwt*fixedDofValues;
```

where `fixedDofs` is a vector of the indicies in $\mathbf{d}$ that are fixed, `fixedDofValues` is a vector of the values that `fixedDofs` are assigned to and `bcwt` is a weighing factor to retain the conditioning of the stiffness matrix (typically $bcwt = trace(\mathbf{K})/N$).

6 Where To Go Next

Hopefully this extremely brief overview of programming simple finite element methods with Matlab has helped bridge the gap between reading the theory of the finite element method and sitting down and writing ones own finite element code. The examples in the Appendix should be looked at and run, but also I would suggest trying to write a simple 1D or 2D finite element code from scratch to really solidify the method in ones head. The examples can then be used as a reference to diminish the struggle. Good Luck!

A Installation of Example Matlab Program

All the functions needed to run the example programs as well as the examples themselves can be found at
`http://www.tam.northwestern.edu/jfc795/Matlab/`
I believe that the following files are required, but if one gets a run error about function not found chances are that I forgot to list it here but it is in one of the Matlab directories at the above web site.

- MeshGenerationsquare_node_array.m: generates an array of nodes in 2D
- MeshGenerationmake_elem.m: generates elements on an array of nodes
- MeshGenerationmsh2mlab.m: reads in a *Gmsh* file
- MeshGenerationplot_mesh.m: plots a finite element mesh
- PostProcessingplot_field.m: plots a finite element field
- quadrature.m: returns various quadrature rules
- lagrange_basis.m: return the shape functions and gradients of the shape functions in the parent coordinate system for various elements

There are many additional files that one might find useful and an interested individual can explore these on there own. These fies should be copied either the directory which contains the example script file or into a directory that is in the Matlab search path.

B Example: Beam Bending Problem

The first example program solves the static bending of a linear elastic beam. The configuration of the problem is shown in Figure 3 and the program can be found at
`http://www.tam.northwestern.edu/jfc795/Matlab/`
`Examples/Static/beam.m`
The exact solution for this problem is as follows

$$\begin{aligned}
\sigma_{11} &= -\frac{P(L-x)y}{I} \\
\sigma 22 &= 0 \\
\sigma 12 &= \frac{P}{2I}(c^2-y^2) \\
u_1 &= -\frac{Py}{6EI}\left\{3\left(L^2-(L-x)^2\right)+(2+\nu)(y^2-c^2)\right)\right\} \\
u_2 &= \frac{Py}{6EI}\left\{3\left((L-x)^3-L^3\right)-\left[(4+5\nu)c^2+3L^2\right]x+3\nu(L-x)y^2\right\}
\end{aligned}$$

This problem can be run with three element types; three node triangle element, a four node quadrilateral element and a nine node quadrilateral element. Also, one can choose between plane strain or plane stress assumption.

```
% beam.m
%
% Solves a linear elastic 2D beam problem ( plane stress or strain )
% with several element types.
%
%         ^ y
%         |
%         ---------------------------------------------
%         |                                           |
%         |                                           |
%         ---------> x                                | 2c
%         |                                           |
%         |                       L                   |
%         ---------------------------------------------
%
% with the boundary following conditions:
%
%       u_x = 0 at (0,0), (0,-c) and (0,c)
%       u_y = 0 at (0,0)
%
%       t_x = y             along the edge x=0
%       t_y = P*(x^2-c^2)   along the edge x=L
%
% ******************************************************************************
%
%      This file and the supporting matlab files can be found at
%             http://www.tam.northwestern.edu/jfc795/Matlab
%
%      by Jack Chessa
%      Northwestern University
```

```
%
% ******************************************************************************

clear
colordef black
state = 0;

% ******************************************************************************
% ***                            I N P U T                                   ***
% ******************************************************************************
tic;
disp('************************************************')
disp('***          S T A R T I N G    R U N       ***')
disp('************************************************')

disp([num2str(toc),'   START'])

% MATERIAL PROPERTIES
E0  = 10e7;  % Young's modulus
nu0 = 0.30;  % Poisson's ratio

% BEAM PROPERTIES
L  = 16;     % length of the beam
c  = 2;      % the distance of the outer fiber of the beam from the mid-line

% MESH PROPERTIES
elemType = 'Q9'; % the element type used in the FEM simulation; 'T3' is for a
                 % three node constant strain triangular element, 'Q4' is for
                 % a four node quadrilateral element, and 'Q9' is for a nine
                 % node quadrilateral element.

numy     = 4;    % the number of elements in the x-direction (beam length)
numx     = 18;   % and in the y-direciton.
plotMesh = 1;    % A flag that if set to 1 plots the initial mesh (to make sure
                 % that the mesh is correct)
% TIP LOAD
P = -1; % the peak magnitude of the traction at the right edge

% STRESS ASSUMPTION
stressState='PLANE_STRESS'; % set to either 'PLANE_STRAIN' or "PLANE_STRESS'
                            % nuff said.

% ******************************************************************************
% ***                    P R E - P R O C E S S I N G                         ***
% ******************************************************************************
I0=2*c^3/3;     % the second polar moment of inertia of the beam cross-section.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% COMPUTE ELASTICITY MATRIX
if ( strcmp(stressState,'PLANE_STRESS') )       % Plane Strain case
  C=E0/(1-nu0^2)*[   1      nu0          0;
                   nu0        1          0;
                     0        0  (1-nu0)/2 ];
else                                            % Plane Strain case
  C=E0/(1+nu0)/(1-2*nu0)*[ 1-nu0      nu0        0;
                             nu0    1-nu0        0;
                               0        0  1/2-nu0 ];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% GENERATE FINITE ELEMENT MESH
%
```

```
% Here we gnerate the finte element mesh (using the approriate elements).
% I won't go into too much detail about how to use these functions.  If
% one is interested one can type - help 'function name' at the matlab comand
% line to find out more about it.
%
% The folowing data structures are used to describe the finite element
% discretization:
%
%   node    - is a matrix of the node coordinates, i.e. node(I,j) -> x_Ij
%   element - is a matrix of element connectivities, i.e. the connectivity
%             of element e is given by > element(e,:) -> [n1 n2 n3 ...];
%
% To apply boundary conditions a description of the boundaries is needed.  To
% accomplish this we use a separate finite element discretization for each
% boundary.  For a 2D problem the boundary discretization is a set of 1D elements.
%
%   rightEdge -  a element connectivity matrix for the right edge
%   leftEdge  -  I'll give you three guesses
%
% These connectivity matricies refer to the node numbers defined in the
% coordinate matrix node.

disp([num2str(toc),'   GENERATING MESH'])
switch elemType
case 'Q4'           % here we generate the mesh of Q4 elements
  nnx=numx+1;
  nny=numy+1;
  node=square_node_array([0 -c],[L -c],[L c],[0 c],nnx,nny);

  inc_u=1;
  inc_v=nnx;
  node_pattern=[ 1 2 nnx+2 nnx+1 ];

  element=make_elem(node_pattern,numx,numy,inc_u,inc_v);

case 'Q9'           % here we generate a mehs of Q9 elements
  nnx=2*numx+1;
  nny=2*numy+1;
  node=square_node_array([0 -c],[L -c],[L c],[0 c],nnx,nny);

  inc_u=2;
  inc_v=2*nnx;
  node_pattern=[ 1 3 2*nnx+3 2*nnx+1 2 nnx+3 2*nnx+2 nnx+1 nnx+2 ];

  element=make_elem(node_pattern,numx,numy,inc_u,inc_v);

otherwise %'T3'    % and last but not least T3 elements
  nnx=numx+1;
  nny=numy+1;
  node=square_node_array([0 -c],[L -c],[L c],[0 c],nnx,nny);

  node_pattern1=[ 1 2 nnx+1 ];
  node_pattern2=[ 2 nnx+2 nnx+1 ];
  inc_u=1;
  inc_v=nnx;

  element=[make_elem(node_pattern1,numx,numy,inc_u,inc_v);
          make_elem(node_pattern2,numx,numy,inc_u,inc_v) ];

end

% DEFINE BOUNDARIES
%   Here we define the boundary discretizations.
uln=nnx*(nny-1)+1;    % upper left node number
urn=nnx*nny;          % upper right node number
lrn=nnx;              % lower right node number
lln=1;                % lower left node number
cln=nnx*(nny-1)/2+1;  % node number at (0,0)
switch elemType
case 'Q9'
  rightEdge=[ lrn:2*nnx:(uln-1); (lrn+2*nnx):2*nnx:urn; (lrn+nnx):2*nnx:urn ]';
  leftEdge =[ uln:-2*nnx:(lrn+1); (uln-2*nnx):-2*nnx:1; (uln-nnx):-2*nnx:1 ]';
  edgeElemType='L3';

otherwise  % same discretizations for Q4 and T3 meshes
  rightEdge=[ lrn:nnx:(uln-1); (lrn+nnx):nnx:urn ]';
  leftEdge =[ uln:-nnx:(lrn+1); (uln-nnx):-nnx:1 ]';
  edgeElemType='L2';

end

% GET NODES ON DISPLACEMENT BOUNDARY
%      Here we get the nodes on the essential boundaries
fixedNodeX=[uln lln cln]';  % a vector of the node numbers which are fixed in
                            % the x direction
fixedNodeY=[cln]';          % a vector of node numbers which are fixed in
                            % the y-direction

uFixed=zeros(size(fixedNodeX));     % a vector of the x-displacement for the nodes
                                    % in fixedNodeX  ( in this case just zeros )
vFixed=zeros(size(fixedNodeY));     % and the y-displacements for fixedNodeY

numnode=size(node,1);    % number of nodes
numelem=size(element,1); % number of elements

% PLOT MESH
if ( plotMesh )  % if plotMesh==1 we will plot the mesh
  clf
  plot_mesh(node,element,elemType,'g.-');
  hold on
  plot_mesh(node,rightEdge,edgeElemType,'bo-');
  plot_mesh(node,leftEdge,edgeElemType,'bo-');
  plot(node(fixedNodeX,1),node(fixedNodeX,2),'r>');
  plot(node(fixedNodeY,1),node(fixedNodeY,2),'r^');
  axis off
  axis([0 L -c c])
  disp('(paused)')
  pause
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DEFINE SYSTEM DATA STRUCTURES
%
% Here we define the system data structures
%   U - is vector of the nodal displacements it is of length 2*numnode. The
%       displacements in the x-direction are in the top half of U and the
%       y-displacements are in the lower half of U, for example the displacement
%       in the y-direction for node number I is at U(I+numnode)
%   f - is the nodal force vector.  It's structure is the same as U,
%       i.e. f(I+numnode) is the force in the y direction at node I
%   K - is the global stiffness matrix and is structured the same as with U and f
%       so that K_IiJj is at K(I+(i-1)*numnode,J+(j-1)*numnode)
disp([num2str(toc),'   INITIALIZING DATA STRUCTURES'])
U=zeros(2*numnode,1);          % nodal displacement vector
```

```
f=zeros(2*numnode,1);          % external load vector
K=sparse(2*numnode,2*numnode); % stiffness matrix

% a vector of indicies that quickly address the x and y portions of the data
% strtuctures so U(xs) returns U_x the nodal x-displacements
xs=1:numnode;                  % x portion of u and v vectors
ys=(numnode+1):2*numnode;      % y portion of u and v vectors

% ******************************************************************************
% ***                          P R O C E S S I N G                           ***
% ******************************************************************************

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% COMPUTE EXTERNAL FORCES
%   integrate the tractions on the left and right edges
disp([num2str(toc),'   COMPUTING EXTERNAL LOADS'])

switch elemType  % define quadrature rule
case 'Q9'
  [W,Q]=quadrature( 4, 'GAUSS', 1 ); %  four point quadrature
otherwise
  [W,Q]=quadrature( 3, 'GAUSS', 1 ); % three point quadrature
end

% RIGHT EDGE
for e=1:size(rightEdge,1) % loop over the elements in the right edge

  sctr=rightEdge(e,:);  % scatter vector for the element
  sctrx=sctr;           % x scatter vector
  sctry=sctrx+numnode;  % y scatter vector

  for q=1:size(W,1)                              % quadrature loop
    pt=Q(q,:);                                   % quadrature point
    wt=W(q);                                     % quadrature weight
    [N,dNdxi]=lagrange_basis(edgeElemType,pt);  % element shape functions
    J0=dNdxi'*node(sctr,:);                      % element Jacobian
    detJ0=norm(J0);                              % determiniat of jacobian

    yPt=N'*node(sctr,2);                 % y coordinate at quadrature point
    fyPt=P*(c^2-yPt^2)/(2*I0);           % y traction at quadrature point
    f(sctry)=f(sctry)+N*fyPt*detJ0*wt;  % scatter force into global force vector

  end % of quadrature loop
end  % of element loop

% LEFT EDGE
for e=1:size(leftEdge,1)    % loop over the elements in the left edge

  sctr=rightEdge(e,:);
  sctrx=sctr;
  sctry=sctrx+numnode;

  for q=1:size(W,1)                              % quadrature loop
    pt=Q(q,:);                                   % quadrature point
    wt=W(q);                                     % quadrature weight
    [N,dNdxi]=lagrange_basis(edgeElemType,pt);  % element shape functions
    J0=dNdxi'*node(sctr,:);                      % element Jacobian
    detJ0=norm(J0);                              % determiniat of jacobian

    yPt=N'*node(sctr,2);
    fyPt=-P*(c^2-yPt^2)/(2*I0);          % y traction at quadrature point
    fxPt=P*L*yPt/I0;                     % x traction at quadrature point
    f(sctry)=f(sctry)+N*fyPt*detJ0*wt;
    f(sctrx)=f(sctrx)+N*fxPt*detJ0*wt;

  end % of quadrature loop
end  % of element loop

% set the force at the nodes on the top and bottom edges to zero (traction free)
% TOP EDGE
topEdgeNodes = find(node(:,2)==c);  % finds nodes on the top edge
f(topEdgeNodes)=0;
f(topEdgeNodes+numnode)=0;

% BOTTOM EDGE
bottomEdgeNodes = find(node(:,2)==-c); % finds nodes on the bottom edge
f(bottomEdgeNodes)=0;
f(bottomEdgeNodes+numnode)=0;

%%%%%%%%%%%%%%%%%%%% COMPUTE STIFFNESS MATRIX %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
disp([num2str(toc),'   COMPUTING STIFFNESS MATRIX'])
switch elemType  % define quadrature rule
case 'Q9'
  [W,Q]=quadrature( 4, 'GAUSS', 2 ); % 4x4 Gaussian quadrature
case 'Q4'
  [W,Q]=quadrature( 2, 'GAUSS', 2 ); % 2x2 Gaussian quadrature
otherwise
  [W,Q]=quadrature( 1, 'TRIANGULAR', 2 ); % 1 point triangural quadrature
end

for e=1:numelem                          % start of element loop

  sctr=element(e,:);           % element scatter vector
  sctrB=[ sctr sctr+numnode ]; % vector that scatters a B matrix

  nn=length(sctr);

  for q=1:size(W,1)                        % quadrature loop
    pt=Q(q,:);                             % quadrature point
    wt=W(q);                               % quadrature weight
    [N,dNdxi]=lagrange_basis(elemType,pt); % element shape functions

    J0=node(sctr,:)'*dNdxi;                % element Jacobian matrix
    invJ0=inv(J0);
    dNdx=dNdxi*invJ0;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % COMPUTE B MATRIX
    %        _                                      _
    %        |   N_1,x  N_2,x  ...     0      0  ... |
    %  B  =  |       0      0  ... N_1,y  N_2,y  ... |
    %        |   N_1,y  N_2,y  ... N_1,x  N_2,x  ... |
    %        -                                      -
    B=zeros(3,2*nn);
    B(1,1:nn)       = dNdx(:,1)';
    B(2,nn+1:2*nn)  = dNdx(:,2)';
    B(3,1:nn)       = dNdx(:,2)';
    B(3,nn+1:2*nn)  = dNdx(:,1)';

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % COMPUTE ELEMENT STIFFNESS AT QUADRATURE POINT
    K(sctrB,sctrB)=K(sctrB,sctrB)+B'*C*B*W(q)*det(J0);

  end  % of quadrature loop
end    % of element loop
%%%%%%%%%%%%%%%%%%%% END OF STIFFNESS MATRIX COMPUTATION %%%%%%%%%%%%%%%%%%%%%%

% APPLY ESSENTIAL BOUNDARY CONDITIONS
disp([num2str(toc),'   APPLYING BOUNDARY CONDITIONS'])
bcwt=mean(diag(K)); % a measure of the average size of an element in K
                   % used to keep the conditioning of the K matrix
udofs=fixedNodeX;           % global indecies of the fixed x displacements
vdofs=fixedNodeY+numnode;   % global indecies of the fixed y displacements

f=f-K(:,udofs)*uFixed;  % modify the force vector
f=f-K(:,vdofs)*vFixed;
f(udofs)=uFixed;
f(vdofs)=vFixed;

K(udofs,:)=0;   % zero out the rows and columns of the K matrix
K(vdofs,:)=0;
K(:,udofs)=0;
K(:,vdofs)=0;
K(udofs,udofs)=bcwt*speye(length(udofs)); % put ones*bcwt on the diagonal
K(vdofs,vdofs)=bcwt*speye(length(vdofs));

% SOLVE SYSTEM
disp([num2str(toc),'   SOLVING SYSTEM'])
U=K\f;

%******************************************************************************
%***                    P O S T  -  P R O C E S S I N G                    ***
%******************************************************************************
%
% Here we plot the stresses and displacements of the solution. As with the
% mesh generation section we don't go into too much detail - use help
% 'function name' to get more details.

disp([num2str(toc),'   POST-PROCESSING'])

dispNorm=L/max(sqrt(U(xs).^2+U(ys).^2));
scaleFact=0.1*dispNorm;
fn=1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PLOT DEFORMED DISPLACEMENT PLOT
figure(fn)
clf
plot_field(node+scaleFact*[U(xs) U(ys)],element,elemType,U(ys));
hold on
plot_mesh(node+scaleFact*[U(xs) U(ys)],element,elemType,'g.-');
plot_mesh(node,element,elemType,'w--');
colorbar
fn=fn+1;
title('DEFORMED DISPLACEMENT IN Y-DIRECTION')

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% COMPUTE STRESS
stress=zeros(numelem,size(element,2),3);

switch elemType  % define quadrature rule
case 'Q9'
  stressPoints=[-1 -1;1 -1;1 1;-1 1;0 -1;1 0;0 1;-1 0;0 0 ];
case 'Q4'
  stressPoints=[-1 -1;1 -1;1 1;-1 1];
otherwise
```

```
  stressPoints=[0 0;1 0;0 1];
end

for e=1:numelem                          % start of element loop

  sctr=element(e,:);
  sctrB=[sctr sctr+numnode];
  nn=length(sctr);

  for q=1:nn
    pt=stressPoints(q,:);                       % stress point
    [N,dNdxi]=lagrange_basis(elemType,pt);    % element shape functions

    J0=node(sctr,:)'*dNdxi;                   % element Jacobian matrix
    invJ0=inv(J0);
    dNdx=dNdxi*invJ0;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % COMPUTE B MATRIX
    B=zeros(3,2*nn);
    B(1,1:nn)       = dNdx(:,1)';
    B(2,nn+1:2*nn)  = dNdx(:,2)';
    B(3,1:nn)       = dNdx(:,2)';
    B(3,nn+1:2*nn)  = dNdx(:,1)';

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % COMPUTE ELEMENT STRAIN AND STRESS AT STRESS POINT
    strain=B*U(sctrB);
    stress(e,q,:)=C*strain;

  end
end   % of element loop

stressComp=1;
figure(fn)
clf
plot_field(node+scaleFact*[U(xs) U(ys)],element,elemType,stress(:,:,stressComp));
hold on
plot_mesh(node+scaleFact*[U(xs) U(ys)],element,elemType,'g.-');
plot_mesh(node,element,elemType,'w--');
colorbar
fn=fn+1;
title('DEFORMED STRESS PLOT, BENDING COMPONENT')
%print(fn,'-djpeg90',['beam_',elemType,'_sigma',num2str(stressComp),'.jpg'])

disp([num2str(toc),'   RUN FINISHED'])
% ***************************************************************************
% ***                    E N D   O F   P R O G R A M                    ***
% ***************************************************************************
disp('************************************************')
disp('***            E N D    O F    R U N        ***')
disp('************************************************')
```

C Example: Modal Analysis of an Atomic Force Microscopy (AFM) Tip

The program presented here is found at
`http://www.tam.northwestern.edu/jfc795/Matlab/Examples /Static/modal_afm.m`
In addition the mesh file `afm.msh` is needed. This mesh file is produced using the GPL program *Gmsh* which is available at
`http://www.geuz.org/gmsh/`
This program is not needed to run this program, only the *.msh file is needed, but it is a very good program for generating finite element meshes. In this example we perform a linear modal analysis of the AFM tip shown in Figure reffig:afm. This involves computing the mass and stiffness matrix and solving the following Eigenvalue problem

$$\left(\mathbf{K} - \omega_n^2 \mathbf{M}\right) \mathbf{a}_n = \mathbf{0} \tag{26}$$

for the natural frequencies ω_n and the corresponding mode shapes $\mathbf{a}_n$. Here the AFM tip is modeled with eight node brick elements and we assume that the feet of the AFM tip are fixed.

```
% modal_afm.m
%
% by Jack Chessa
% Northwestern University
%
clear
colordef black
state = 0;

%*******************************************************************************
%***                              I N P U T                                  ***
%*******************************************************************************
tic;
disp('************************************************')
disp('***          S T A R T I N G   R U N       ***')
disp('************************************************')

disp([num2str(toc),'   START'])

% MATERIAL PROPERTIES
E0  = 160;      % Youngs modulus in GPa
nu0 = 0.27;     % Poisson ratio
rho = 2.330e-9; % density in 10e12 Kg/m^3

% MESH PARAMETERS
quadType='GAUSS';
quadOrder=2;

% GMSH PARAMETERS
fileName='afm.msh';
domainID=50;
```

```
fixedID=51;
topID=52;

% EIGENPROBELM SOLUTION PARAMETERS
numberOfModes=8;   % number of modes to compute
consistentMass=0;  % use a consistent mass matrix
fixedBC=1;         % use fixed or free bcs

%******************************************************************************
%***                   P R E - P R O C E S S I N G                          ***
%******************************************************************************

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% READ GMSH FILE
disp([num2str(toc),'   READING GMSH FILE'])
[node,elements,elemType]=msh2mlab(fileName);
[node,elements]=remove_free_nodes(node,elements);
element=elements{domainID};
element=brickcheck(node,element,1);
if ( fixedBC )
  fixedEdge=elements{fixedID};
else
  fixedEdge=[];
end
topSurface=elements{topID};

plot_mesh(node,element,elemType{domainID},'r-')
disp([num2str(toc),'   INITIALIZING DATA STRUCTURES'])
numnode=size(node,1);    % number of nodes
numelem=size(element,1); % number of elements

% GET NODES ON DISPLACEMENT BOUNDARY
fixedNodeX=unique(fixedEdge);
fixedNodeY=fixedNodeX;
fixedNodeZ=fixedNodeX;

uFixed=zeros(size(fixedNodeX));  % displacement for fixed nodes
vFixed=zeros(size(fixedNodeY));
wFixed=zeros(size(fixedNodeZ));


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% COMPUTE COMPLIANCE MATRIX
C=zeros(6,6);
C(1:3,1:3)=E0/(1+nu0)/(1-2*nu0)*[ 1-nu0      nu0      nu0;
                                    nu0    1-nu0      nu0;
                                    nu0      nu0    1-nu0 ];
C(4:6,4:6)=E0/(1+nu0)*eye(3);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% DEFINE SYSTEM DATA STRUCTURES
K=sparse(3*numnode,3*numnode); % stiffness matrix
if ( consistentMass)
  M=sparse(3*numnode,3*numnode); % mass matrix
else
  M=zeros(3*numnode,1); % mass vector
end

%******************************************************************************
%***                         P R O C E S S I N G                            ***
%******************************************************************************
```

```
%%%%%%%%%%%%%%%%%%%%% COMPUTE SYSTEM MATRICIES %%%%%%%%%%%%%%%%%%%%%%%%%%%%
disp([num2str(toc),'   COMPUTING STIFFNESS AND MASS MATRIX'])
[W,Q]=quadrature(quadOrder,quadType,3); % define quadrature rule
et=elemType{domainID};
nn=size(element,2);
for e=1:numelem                          % start of element loop

  sctr=element(e,:);           % element scatter vector
  sctrB0=[ sctr sctr+numnode sctr+2*numnode ]; %  scatters a B matrix

  for q=1:size(W,1)                          % quadrature loop
    pt=Q(q,:);                               % quadrature point
    wt=W(q);                                 % quadrature weight
    [N,dNdxi]=lagrange_basis(et,pt);       % element shape functions

    J0=node(sctr,:)'*dNdxi;                  % element Jacobian matrix
    invJ0=inv(J0);
    dNdx=dNdxi*invJ0;
    detJ0=det(J0);

    if (detJ0 <= 0)
      disp(['ERROR: NEGATIVE JACOBIAN IN ELEMENT ',num2str(e)]);
    end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % COMPUTE B MATRIX
    B0=zeros(6,3*nn);
    B0(1,1:nn)        = dNdx(:,1)';
    B0(2,nn+1:2*nn)   = dNdx(:,2)';
    B0(3,2*nn+1:3*nn) = dNdx(:,3)';

    B0(4,2*nn+1:3*nn) = dNdx(:,2)';
    B0(4,nn+1:2*nn)   = dNdx(:,3)';

    B0(5,1:nn)        = dNdx(:,3)';
    B0(5,2*nn+1:3*nn) = dNdx(:,1)';

    B0(6,nn+1:2*nn)   = dNdx(:,1)';
    B0(6,1:nn)        = dNdx(:,2)';

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % COMPUTE ELEMENT STIFFNESS AT QUADRATURE POINT
    K(sctrB0,sctrB0)=K(sctrB0,sctrB0)+B0'*C*B0*wt*detJ0;

    % COMPUTE ELEMENT MASS AT QUADRATURE POINT
    mQPt=N*rho*N'*wt*detJ0;
    if ( ~consistentMass )
      mQPt=sum(mQPt)';
      M(sctr)           = M(sctr)+mQPt;
      M(sctr+numnode)   = M(sctr+numnode)+mQPt;
      M(sctr+2*numnode) = M(sctr+2*numnode)+mQPt;
    else
      M(sctr,sctr)                        = M(sctr,sctr)+mQPt;
      M(sctr+numnode,sctr+numnode)        = M(sctr+numnode,sctr+numnode)+mQPt;
      M(sctr+2*numnode,sctr+2*numnode)    = M(sctr+2*numnode,sctr+2*numnode)+mQPt;
    end
  end                                        % of quadrature loop
end     % of element loop
%%%%%%%%%%%%%%%%%%% END OF SYSTEM MATRIX COMPUTATION %%%%%%%%%%%%%%%%%%%%%

% ELIMINATE FIXED DOFS FROM EIGENVALUE COMUTATION
disp([num2str(toc),'   FINDING ACTIVE DOFS'])
```

```
activeDof=setdiff([1:numnode]',[fixedNodeX;fixedNodeY;fixedNodeZ]);
activeDof=[activeDof;activeDof+numnode;activeDof+2*numnode];

% SOLVE SYSTEM
disp([num2str(toc),'   SOLVING EIGEN PROBLEM'])
if ( consistentMass )
  [modeShape,freq]=eigs(K(activeDof,activeDof),M(activeDof,activeDof),...
                              numberOfModes,0);
else
  Minv=spdiags(1./M,0,3*numnode,3*numnode);
  K=Minv*K;
  [modeShape,freq]=eigs(K(activeDof,activeDof),numberOfModes,0);
end
freq=diag(freq)/(2*pi);  % frequency in kHz

%******************************************************************************
%***                    P O S T  -  P R O C E S S I N G                     ***
%******************************************************************************
disp([num2str(toc),'   POST-PROCESSING'])
disp(['THE MODE FREQUENCIES ARE:'])
for m=1:length(freq)
  disp(['   MODE: ',num2str(m),' ',num2str(freq(m))])

  % PLOT MODE SHAPE
  figure(m); clf;
  U=zeros(numnode,1);
  U(activeDof)=modeShape(:,m);
  scaleFactor=20/max(abs(U));
  plot_field(node+[U(1:numnode) U(numnode+1:2*numnode)
          U(2*numnode+1:3*numnode)]*scaleFactor,topSurface,elemType{topID},...
          ones(3*numnode,1));
  hold on
  plot_mesh(node+[U(1:numnode) U(numnode+1:2*numnode)
         U(2*numnode+1:3*numnode)]*scaleFactor,topSurface,elemType{topID},'k-');
  plot_mesh(node,topSurface,elemType{topID},'r-');
  title(['MODE ',num2str(m),', FREQUENCY = ',num2str(freq(m)),' [kHz]'])
  view(37,36)
  axis off
  print(m, '-djpeg90', ['afm_mode_',num2str(m),'.jpg']);

end

% ANIMATE MODE
nCycles=5; % number of cycles to animate
fpc=10;    % frames per cycle
fact=sin(linspace(0,2*pi,fpc));
m=input('What mode would you like to animate (type 0 to exit) ');
while ( m~=0 )

  U=zeros(numnode,1);
  U(activeDof)=modeShape(:,m);
  wt=20/max(abs(U));
  for i=1:fpc
    scaleFactor=fact(i)*wt;
    figure(length(freq+1));
    clf;
    plot_field(node+[U(1:numnode) U(numnode+1:2*numnode)
            U(2*numnode+1:3*numnode)]*scaleFactor,topSurface,elemType{topID},...
                    ones(3*numnode,1));
    hold on
    plot_mesh(node+[U(1:numnode) U(numnode+1:2*numnode)
           U(2*numnode+1:3*numnode)]*scaleFactor,topSurface,elemType{topID},'k-');
    plot_mesh(node,topSurface,elemType{topID},'w-');
    hold on
    view(37,36)
    axis([70 240 30 160 -10 10])
    title(['MODE ',num2str(m),', FREQUENCY = ',num2str(freq(m)),' [kHz]'])
    axis off
    film(i)=getframe;
  end

  movie(film,nCycles);

  m=input('What mode would you like to animate (type 0 to exit) ');
  if ( m > length(freq) )
    disp(['mode must be less than ',num2str(length(freq))])
  end
end

disp([num2str(toc),'   RUN FINISHED'])
% ***************************************************************************
% ***                     E N D   O F   P R O G R A M                     ***
% ***************************************************************************
disp('************************************************')
disp('***           E N D    O F    R U N        ***')
disp('************************************************')

% compute uexact
```

D Common Matlab Functions

Here is a quick list of some built in Matlab functions. These discriptions are availible by using the help function in Matlab.

```
>> help

HELP topics:

matlab/general       -  General purpose commands.
matlab/ops           -  Operators and special characters.
matlab/lang          -  Language constructs and debugging.
matlab/elmat         -  Elementary matrices and matrix manipulation.
matlab/specmat       -  Specialized matrices.
matlab/elfun         -  Elementary math functions.
matlab/specfun       -  Specialized math functions.
matlab/matfun        -  Matrix functions - numerical linear algebra.
matlab/datafun       -  Data analysis and Fourier transform functions.
matlab/polyfun       -  Polynomial and interpolation functions.
matlab/funfun        -  Function functions - nonlinear numerical methods.
matlab/sparfun       -  Sparse matrix functions.
matlab/plotxy        -  Two dimensional graphics.
matlab/plotxyz       -  Three dimensional graphics.
matlab/graphics      -  General purpose graphics functions.
matlab/color         -  Color control and lighting model functions.
matlab/sounds        -  Sound processing functions.
matlab/strfun        -  Character string functions.
matlab/iofun         -  Low-level file I/O functions.
matlab/demos         -  The MATLAB Expo and other demonstrations.
toolbox/chem         -  Chemometrics Toolbox
toolbox/control      -  Control System Toolbox.
fdident/fdident      -  Frequency Domain System Identification Toolbox
fdident/fddemos      -  Demonstrations for the FDIDENT Toolbox
toolbox/hispec       -  Hi-Spec Toolbox
toolbox/ident        -  System Identification Toolbox.
toolbox/images       -  Image Processing Toolbox.
toolbox/local        -  Local function library.
toolbox/mmle3        -  MMLE3 Identification Toolbox.
mpc/mpccmds          -  Model Predictive Control Toolbox
mpc/mpcdemos         -  Model Predictive Control Toolbox
mutools/commands     -  Mu-Analysis and Synthesis Toolbox.: Commands directory
```

```
mutools/subs        -  Mu-Analysis and Synthesis Toolbox -- Supplement
toolbox/ncd         -  Nonlinear Control Design Toolbox.
nnet/nnet           -  Neural Network Toolbox.
nnet/nndemos        -  Neural Network Demonstrations and Applications.
toolbox/optim       -  Optimization Toolbox.
toolbox/robust      -  Robust Control Toolbox.
toolbox/signal      -  Signal Processing Toolbox.
toolbox/splines     -  Spline Toolbox.
toolbox/stats       -  Statistics Toolbox.
toolbox/symbolic    -  Symbolic Math Toolbox.
toolbox/wavbox      - (No table of contents file)
simulink/simulink   -  SIMULINK model analysis and construction functions.
simulink/blocks     -  SIMULINK block library.
simulink/simdemos   -  SIMULINK demonstrations and samples.
toolbox/codegen     -  Real-Time Workshop

For more help on directory/topic, type "help topic".

>> help elmat

 Elementary matrices and matrix manipulation.

 Elementary matrices.
   zeros       - Zeros matrix.
   ones        - Ones matrix.
   eye         - Identity matrix.
   rand        - Uniformly distributed random numbers.
   randn       - Normally distributed random numbers.
   linspace    - Linearly spaced vector.
   logspace    - Logarithmically spaced vector.
   meshgrid    - X and Y arrays for 3-D plots.
   :           - Regularly spaced vector.

 Special variables and constants.
   ans         - Most recent answer.
   eps         - Floating point relative accuracy.
   realmax     - Largest floating point number.
   realmin     - Smallest positive floating point number.
   pi          - 3.1415926535897....
   i, j        - Imaginary unit.
   inf         - Infinity.
```

```
    NaN          - Not-a-Number.
    flops        - Count of floating point operations.
    nargin       - Number of function input arguments.
    nargout      - Number of function output arguments.
    computer     - Computer type.
    isieee       - True for computers with IEEE arithmetic.
    isstudent    - True for the Student Edition.
    why          - Succinct answer.
    version      - MATLAB version number.

  Time and dates.
    clock        - Wall clock.
    cputime      - Elapsed CPU time.
    date         - Calendar.
    etime        - Elapsed time function.
    tic, toc     - Stopwatch timer functions.

  Matrix manipulation.
    diag         - Create or extract diagonals.
    fliplr       - Flip matrix in the left/right direction.
    flipud       - Flip matrix in the up/down direction.
    reshape      - Change size.
    rot90        - Rotate matrix 90 degrees.
    tril         - Extract lower triangular part.
    triu         - Extract upper triangular part.
    :            - Index into matrix, rearrange matrix.

>> help specmat

  Specialized matrices.

    compan       - Companion matrix.
    gallery      - Several small test matrices.
    hadamard     - Hadamard matrix.
    hankel       - Hankel matrix.
    hilb         - Hilbert matrix.
    invhilb      - Inverse Hilbert matrix.
    kron         - Kronecker tensor product.
    magic        - Magic square.
    pascal       - Pascal matrix.
    rosser       - Classic symmetric eigenvalue test problem.
    toeplitz    - Toeplitz matrix.
    vander      - Vandermonde matrix.
    wilkinson   - Wilkinson's eigenvalue test matrix.

>> help elfun

 Elementary math functions.

 Trigonometric.
    sin         - Sine.
    sinh        - Hyperbolic sine.
    asin        - Inverse sine.
    asinh       - Inverse hyperbolic sine.
    cos         - Cosine.
    cosh        - Hyperbolic cosine.
    acos        - Inverse cosine.
    acosh       - Inverse hyperbolic cosine.
    tan         - Tangent.
    tanh        - Hyperbolic tangent.
    atan        - Inverse tangent.
    atan2       - Four quadrant inverse tangent.
    atanh       - Inverse hyperbolic tangent.
    sec         - Secant.
    sech        - Hyperbolic secant.
    asec        - Inverse secant.
    asech       - Inverse hyperbolic secant.
    csc         - Cosecant.
    csch        - Hyperbolic cosecant.
    acsc        - Inverse cosecant.
    acsch       - Inverse hyperbolic cosecant.
    cot         - Cotangent.
    coth        - Hyperbolic cotangent.
    acot        - Inverse cotangent.
    acoth       - Inverse hyperbolic cotangent.

 Exponential.
    exp         - Exponential.
    log         - Natural logarithm.
    log10       - Common logarithm.
    sqrt        - Square root.
  Complex.
    abs         - Absolute value.
    angle       - Phase angle.
    conj        - Complex conjugate.
    imag        - Complex imaginary part.
    real        - Complex real part.

  Numeric.
    fix         - Round towards zero.
    floor       - Round towards minus infinity.
    ceil        - Round towards plus infinity.
    round       - Round towards nearest integer.
    rem         - Remainder after division.
    sign        - Signum function.

 >> help specfun

  Specialized math functions.

    besselj     - Bessel function of the first kind.
    bessely     - Bessel function of the second kind.
    besseli     - Modified Bessel function of the first kind.
    besselk     - Modified Bessel function of the second kind.
    beta        - Beta function.
    betainc     - Incomplete beta function.
    betaln      - Logarithm of beta function.
    ellipj      - Jacobi elliptic functions.
    ellipke     - Complete elliptic integral.
    erf         - Error function.
    erfc        - Complementary error function.
    erfcx       - Scaled complementary error function.
    erfinv      - Inverse error function.
    expint      - Exponential integral function.
    gamma       - Gamma function.
    gcd         - Greatest common divisor.
    gammainc    - Incomplete gamma function.
    lcm         - Least common multiple.
    legendre    - Associated Legendre function.
    gammaln     - Logarithm of gamma function.
    log2        - Dissect floating point numbers.
    pow2        - Scale floating point numbers.
```

```
    rat          - Rational approximation.
    rats         - Rational output.
    cart2sph     - Transform from Cartesian to spherical coordinates.
    cart2pol     - Transform from Cartesian to polar coordinates.
    pol2cart     - Transform from polar to Cartesian coordinates.
    sph2cart     - Transform from spherical to Cartesian coordinates.

>> help matfun

  Matrix functions - numerical linear algebra.

  Matrix analysis.
    cond         - Matrix condition number.
    norm         - Matrix or vector norm.
    rcond        - LINPACK reciprocal condition estimator.
    rank         - Number of linearly independent rows or columns.
    det          - Determinant.
    trace        - Sum of diagonal elements.
    null         - Null space.
    orth         - Orthogonalization.
    rref         - Reduced row echelon form.

  Linear equations.
    \ and /      - Linear equation solution; use "help slash".
    chol         - Cholesky factorization.
    lu           - Factors from Gaussian elimination.
    inv          - Matrix inverse.
    qr           - Orthogonal-triangular decomposition.
    qrdelete     - Delete a column from the QR factorization.
    qrinsert     - Insert a column in the QR factorization.
    nnls         - Non-negative least-squares.
    pinv         - Pseudoinverse.
    lscov        - Least squares in the presence of known covariance.

  Eigenvalues and singular values.
    eig          - Eigenvalues and eigenvectors.
    poly         - Characteristic polynomial.
    polyeig      - Polynomial eigenvalue problem.
    hess         - Hessenberg form.
    qz           - Generalized eigenvalues.
    rsf2csf      - Real block diagonal form to complex diagonal form.
    cdf2rdf     - Complex diagonal form to real block diagonal form.
    schur       - Schur decomposition.
    balance     - Diagonal scaling to improve eigenvalue accuracy.
    svd         - Singular value decomposition.

  Matrix functions.
    expm        - Matrix exponential.
    expm1       - M-file implementation of expm.
    expm2       - Matrix exponential via Taylor series.
    expm3       - Matrix exponential via eigenvalues and eigenvectors.
    logm        - Matrix logarithm.
    sqrtm       - Matrix square root.
    funm        - Evaluate general matrix function.

>> help general

  General purpose commands.
  MATLAB Toolbox  Version 4.2a 25-Jul-94

  Managing commands and functions.
    help        - On-line documentation.
    doc         - Load hypertext documentation.
    what        - Directory listing of M-, MAT- and MEX-files.
    type        - List M-file.
    lookfor     - Keyword search through the HELP entries.
    which       - Locate functions and files.
    demo        - Run demos.
    path        - Control MATLAB's search path.

  Managing variables and the workspace.
    who         - List current variables.
    whos        - List current variables, long form.
    load        - Retrieve variables from disk.
    save        - Save workspace variables to disk.
    clear       - Clear variables and functions from memory.
    pack        - Consolidate workspace memory.
    size        - Size of matrix.
    length      - Length of vector.
    disp        - Display matrix or text.

  Working with files and the operating system.
```

```
    cd          - Change current working directory.
    dir         - Directory listing.
    delete      - Delete file.
    getenv      - Get environment value.
    !           - Execute operating system command.
    unix        - Execute operating system command & return result.
    diary       - Save text of MATLAB session.

  Controlling the command window.
    cedit       - Set command line edit/recall facility parameters.
    clc         - Clear command window.
    home        - Send cursor home.
    format      - Set output format.
    echo        - Echo commands inside script files.
    more        - Control paged output in command window.

  Starting and quitting from MATLAB.
    quit        - Terminate MATLAB.
    startup     - M-file executed when MATLAB is invoked.
    matlabrc    - Master startup M-file.

  General information.
    info        - Information about MATLAB and The MathWorks, Inc.
    subscribe   - Become subscribing user of MATLAB.
    hostid      - MATLAB server host identification number.
    whatsnew    - Information about new features not yet documented.
    ver         - MATLAB, SIMULINK, and TOOLBOX version information.

>> help funfun

  Function functions - nonlinear numerical methods.

    ode23       - Solve differential equations, low order method.
    ode23p      - Solve and plot solutions.
    ode45       - Solve differential equations, high order method.
    quad        - Numerically evaluate integral, low order method.
    quad8       - Numerically evaluate integral, high order method.
    fmin        - Minimize function of one variable.
    fmins       - Minimize function of several variables.
    fzero       - Find zero of function of one variable.
    fplot       - Plot function.
```

```
 See also The Optimization Toolbox, which has a comprehensive
 set of function functions for optimizing and minimizing functions.

>> help polyfun

 Polynomial and interpolation functions.

 Polynomials.
   roots       - Find polynomial roots.
   poly        - Construct polynomial with specified roots.
   polyval     - Evaluate polynomial.
   polyvalm    - Evaluate polynomial with matrix argument.
   residue     - Partial-fraction expansion (residues).
   polyfit     - Fit polynomial to data.
   polyder     - Differentiate polynomial.
   conv        - Multiply polynomials.
   deconv      - Divide polynomials.

 Data interpolation.
   interp1     - 1-D interpolation (1-D table lookup).
   interp2     - 2-D interpolation (2-D table lookup).
   interpft    - 1-D interpolation using FFT method.
   griddata    - Data gridding.

 Spline interpolation.
   spline      - Cubic spline data interpolation.
   ppval       - Evaluate piecewise polynomial.

>> help ops

 Operators and special characters.

  Char    Name                         HELP topic

   +      Plus                         arith
   -      Minus                        arith
   *      Matrix multiplication        arith
   .*     Array multiplication         arith
   ^      Matrix power                 arith
   .^     Array power                  arith
   \      Backslash or left division   slash
   /      Slash or right division      slash
   ./     Array division               slash
   kron   Kronecker tensor product     kron

   :      Colon                        colon

   ( )    Parentheses                  paren
   [ ]    Brackets                     paren

   .      Decimal point                punct
   ..     Parent directory             punct
   ...    Continuation                 punct
   ,      Comma                        punct
   ;      Semicolon                    punct
   %      Comment                      punct
   !      Exclamation point            punct
   '      Transpose and quote          punct
   =      Assignment                   punct

   ==     Equality                     relop
   <,> Relational operators         relop
   & Logical AND                 relop
   |      Logical OR                   relop
   ~      Logical NOT                  relop
   xor    Logical EXCLUSIVE OR         xor

 Logical characteristics.
   exist       - Check if variables or functions are defined.
   any         - True if any element of vector is true.
   all         - True if all elements of vector are true.
   find        - Find indices of non-zero elements.
   isnan       - True for Not-A-Number.
   isinf       - True for infinite elements.
   finite      - True for finite elements.
   isempty     - True for empty matrix.
   isreal      - True for real matrix.
   issparse    - True for sparse matrix.
   isstr       - True for text string.
   isglobal    - True for global variables.
```

```
>> help lang

 Language constructs and debugging.

 MATLAB as a programming language.
   script      - About MATLAB scripts and M-files.
   function    - Add new function.
   eval        - Execute string with MATLAB expression.
   feval       - Execute function specified by string.
   global      - Define global variable.
   nargchk     - Validate number of input arguments.
   lasterr     - Last error message.

 Control flow.
   if          - Conditionally execute statements.
   else        - Used with IF.
   elseif      - Used with IF.
   end         - Terminate the scope of FOR, WHILE and IF statements.
   for         - Repeat statements a specific number of times.
   while       - Repeat statements an indefinite number of times.
   break       - Terminate execution of loop.
   return      - Return to invoking function.
   error       - Display message and abort function.

 Interactive input.
   input       - Prompt for user input.
   keyboard    - Invoke keyboard as if it were a Script-file.
   menu        - Generate menu of choices for user input.
   pause       - Wait for user response.
   uimenu      - Create user interface menu.
   uicontrol   - Create user interface control.

 Debugging commands.
   dbstop     - Set breakpoint.
   dbclear    - Remove breakpoint.
   dbcont     - Resume execution.
   dbdown     - Change local workspace context.
   dbstack    - List who called whom.
   dbstatus   - List all breakpoints.
   dbstep     - Execute one or more lines.
   dbtype     - List M-file with line numbers.
   dbup       - Change local workspace context.
   dbquit     - Quit debug mode.
   mexdebug   - Debug MEX-files.

>> help plotxy
 Two dimensional graphics.

 Elementary X-Y graphs.
   plot      - Linear plot.
   loglog    - Log-log scale plot.
   semilogx  - Semi-log scale plot.
   semilogy  - Semi-log scale plot.
   fill      - Draw filled 2-D polygons.

 Specialized X-Y graphs.
   polar     - Polar coordinate plot.
   bar       - Bar graph.
   stem      - Discrete sequence or "stem" plot.
   stairs    - Stairstep plot.
   errorbar  - Error bar plot.
   hist      - Histogram plot.
   rose      - Angle histogram plot.
   compass   - Compass plot.
   feather   - Feather plot.
   fplot     - Plot function.
   comet     - Comet-like trajectory.

 Graph annotation.
   title     - Graph title.
   xlabel    - X-axis label.
   ylabel    - Y-axis label.
   text      - Text annotation.
   gtext     - Mouse placement of text.
   grid      - Grid lines.

 See also PLOTXYZ, GRAPHICS.

>> help plotxyz

 Three dimensional graphics.
```

```
Line and area fill commands.
  plot3      - Plot lines and points in 3-D space.
  fill3      - Draw filled 3-D polygons in 3-D space.
  comet3     - 3-D comet-like trajectories.

Contour and other 2-D plots of 3-D data.
  contour    - Contour plot.
  contour3   - 3-D contour plot.
  clabel     - Contour plot elevation labels.
  contourc   - Contour plot computation (used by contour).
  pcolor     - Pseudocolor (checkerboard) plot.
  quiver     - Quiver plot.

Surface and mesh plots.
  mesh       - 3-D mesh surface.
  meshc      - Combination mesh/contour plot.
  meshz      - 3-D Mesh with zero plane.
  surf       - 3-D shaded surface.
  surfc      - Combination surf/contour plot.
  surfl      - 3-D shaded surface with lighting.
  waterfall  - Waterfall plot.

Volume visualization.
  slice      - Volumetric visualization plots.

Graph appearance.
  view       - 3-D graph viewpoint specification.
  viewmtx    - View transformation matrices.
  hidden     - Mesh hidden line removal mode.
  shading    - Color shading mode.
  axis       - Axis scaling and appearance.
  caxis      - Pseudocolor axis scaling.
  colormap   - Color look-up table.

Graph annotation.
  title      - Graph title.
  xlabel     - X-axis label.
  ylabel     - Y-axis label.
  zlabel     - Z-axis label for 3-D plots.
  text       - Text annotation.
```

```
    gtext      - Mouse placement of text.
    grid       - Grid lines.

  3-D objects.
    cylinder   - Generate cylinder.
    sphere     - Generate sphere.

  See also COLOR, PLOTXY, GRAPHICS.

>> help strfun

  Character string functions.

  General.
    strings     - About character strings in MATLAB.
    abs         - Convert string to numeric values.
    setstr      - Convert numeric values to string.
    isstr       - True for string.
    blanks      - String of blanks.
    deblank     - Remove trailing blanks.
    str2mat     - Form text matrix from individual strings.
    eval        - Execute string with MATLAB expression.

  String comparison.
    strcmp      - Compare strings.
    findstr     - Find one string within another.
    upper       - Convert string to uppercase.
    lower       - Convert string to lowercase.
    isletter    - True for letters of the alphabet.
    isspace     - True for white space characters.
    strrep      - Replace a string with another.
    strtok      - Find a token in a string.

  String to number conversion.
    num2str     - Convert number to string.
    int2str     - Convert integer to string.
    str2num     - Convert string to number.
    mat2str     - Convert matrix to string.
    sprintf     - Convert number to string under format control.
    sscanf      - Convert string to number under format control.
```

```
 Hexadecimal to number conversion.
   hex2num     - Convert hex string to IEEE floating point number.
   hex2dec     - Convert hex string to decimal integer.
   dec2hex     - Convert decimal integer to hex string.
```

Also the MathWorks web site has a lot of good tutorials, examples and reference documentation.

`http://www.mathworks.com`

A good tutorial is at

```
http://www.mathworks.com/access/helpdesk/help/techdoc/
                                 learn_matlab/learn_matlab.shtml
```

List of Figures

1 A simple finite element mesh of triangular elements 42

2 An example of a element force vector $\mathbf{f}^e$ *scattered* into a global force vector $\mathbf{f}$ 43

3 Diagram of beam used in beam bending example. The following displacement boundary conditions are applied: $u_x = 0$ at the points $(0, \pm c)$ and (0,0), $u_y = 0$ at $(0, 0)$. The following traction boundary conditions are used $t_x = y$ on $x = 0$ and $t_y = P(x^2 - c^2)$ on $x = L$. 44

4 AFM tip modeled in modal analysis example 45

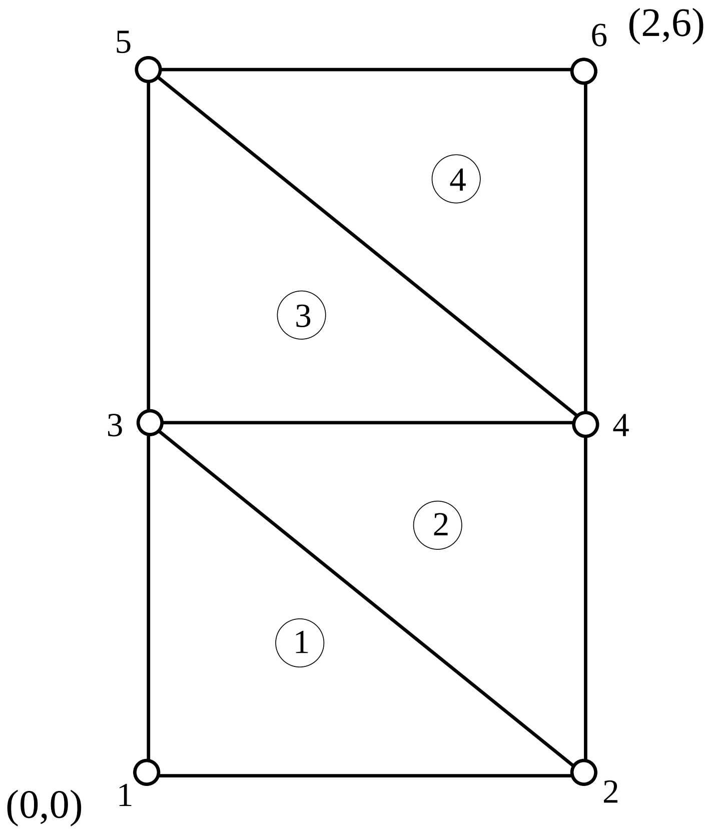

Figure 1: A simple finite element mesh of triangular elements

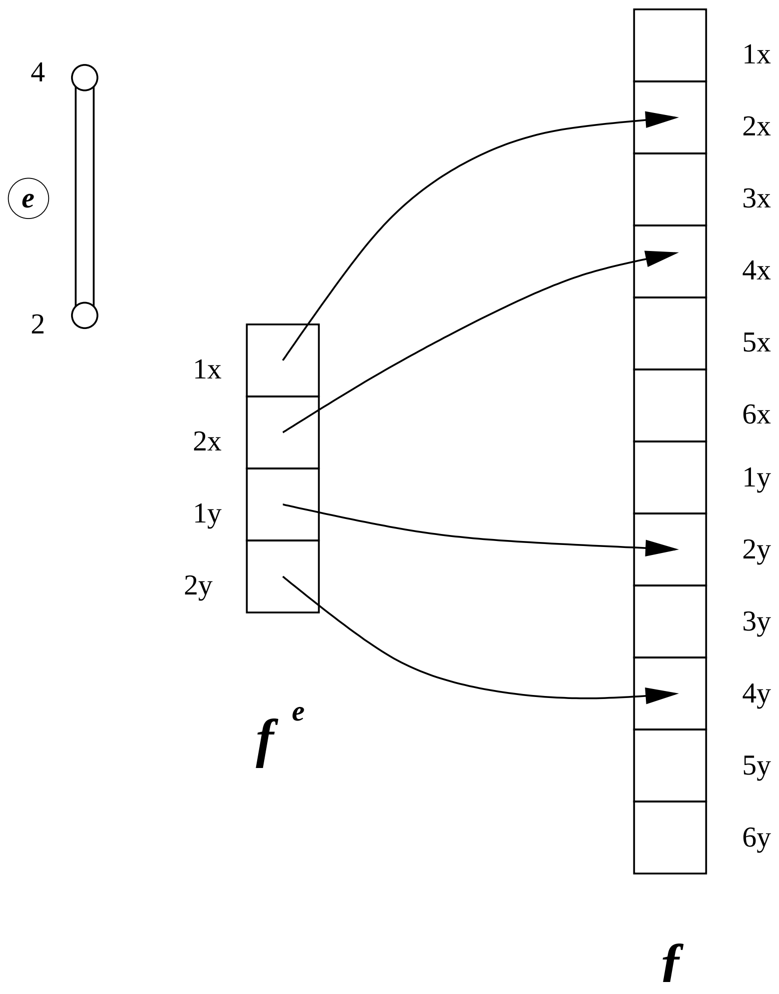

Figure 2: An example of a element force vector $\mathbf{f}^e$ *scattered* into a global force vector $\mathbf{f}$

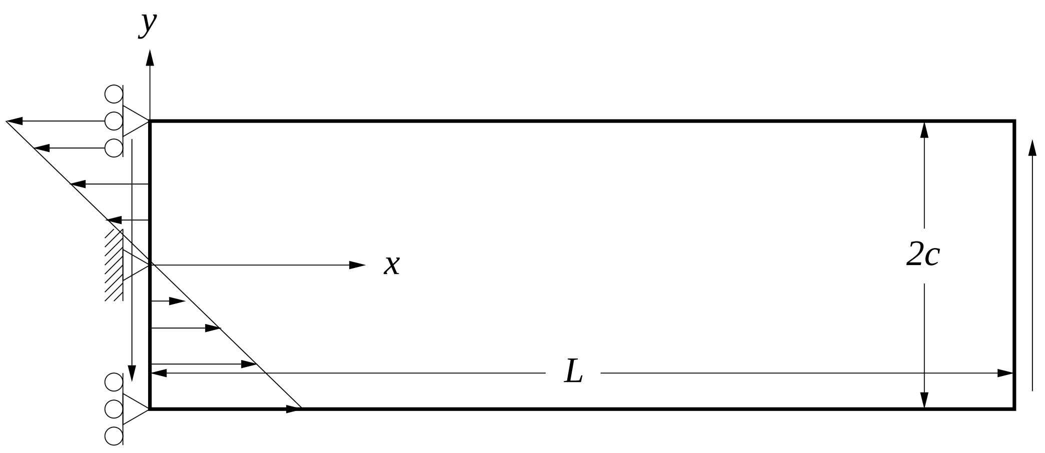

Figure 3: Diagram of beam used in beam bending example. The following displacement boundary conditions are applied: $u_x = 0$ at the points $(0, \pm c)$ and (0,0), $u_y = 0$ at $(0, 0)$. The following traction boundary conditions are used $t_x = y$ on $x = 0$ and $t_y = P(x^2 - c^2)$ on $x = L$.

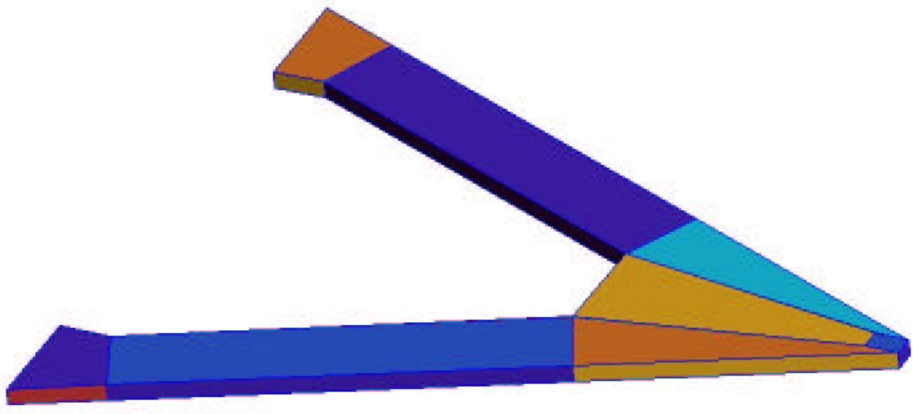

Figure 4: AFM tip modeled in modal analysis example