Introduction to the finite element method for solid mechanics

Alex P. Martinez

March 10, 2011

Contents

1 Introduction

The finite element method (FEM) or finite element analysis (FEA) is a method for solving partial differential equations (PDEs) using trial function methods with point and piecewise discritization. It is used in cases where an analytical solution is either difficult or impossible to obtain with any other known method. Most practical problems in engineering due to their geometric and material complexity require FEA for a satisfactory analysis. Nevertheless, this approach introduces both approximation and round-off error to the solution. This whitepaper assumes some basic knowledge of FEA. The finite element formulation is based on [1].

2 Finite element formulation

This formulation is limited to 2D stress problems. The finite element method uses piecewise discretization and approximation of relations between displacement and force applied in each element. This is given by an element stiffness matrix that by definition relates the displacements and the forces applied to each element:

$$\mathbf{K^e U^e = F^e} \tag{1}$$

The discretization is done by a suitable meshing algorithm depending on the element type used. The element stiffness matrices are then combined based on the mapping between the element and domain displacements into a global stiffness matrix:

$$\mathbf{K^g U^g = F^g} \tag{2}$$

This is referred to as assembly. The system is then solved using a suitable linear solver algorithm. The solution can be carried out faster than a problem with a random matrix of the same size since the global stiffness matrix is symmetric and positive definite. It also usually has a small bandwidth.

The element stiffness matrix is derived as follows using the total energy of the element. The energy of an element with an applied force is as follows:

$$\mathbf{X = \frac{1}{2}\int_{V_e} \varepsilon^T \sigma dV - U_e^T F_e} \tag{3}$$

In the elastic region of a material, the stress is related to the strain by a matrix as follows:

$$\varepsilon = \mathbf{B U_e} \tag{4}$$

The matrix is obtained from the displacement in the element that depends on the nodal displacements, interpolated using shape functions (together being trial functions). The stress is related to the strain in the following way:

$$\sigma = \mathbf{D}\varepsilon \tag{5}$$

It depends on the material properties and the kind of loading, for this formulation we are assuming plane stress. The equations used in 2D plane stress are derived in [2]. Combining the above equations yields:

$$\mathbf{X} = \frac{\mathbf{1}}{\mathbf{2}}\mathbf{U_e^T}\left(\int_{\mathbf{V_e}} \mathbf{B^T D B dV}\right)\mathbf{U_e} - \mathbf{U_e^T F_e} \tag{6}$$

Notice that the displacements are fixed with respect to the integral and can therefore be taken outside of it. Using the Rayleigh-Ritz variational approach, the total energy can be extremized to yield the force as a function of the nodal displacements with the following theorems [1]:

$$\frac{\partial}{\partial\omega}\left(\frac{\mathbf{1}}{\mathbf{2}}\omega^{\mathbf{T}}\mathbf{Q}\omega\right) = \mathbf{Q}\omega \tag{7}$$

$$\frac{\partial}{\partial\omega}\left(\omega^{\mathbf{T}}\mathbf{P}\right) = \mathbf{P} \tag{8}$$

Using the above theorems, extremizing the energy and equating to zero yields:

$$\mathbf{0} = \int_{\mathbf{V_e}} \mathbf{B^T D B dV} - \mathbf{F_e} \tag{9}$$

Notice that in reality the energy should be zero but since we are using trial functions the energy equation will not be equal to zero. However, extremizing it to find the minimum produces an element stiffness matrix that produces the lowest energy possible with the variational approach.

Therefore:

$$\mathbf{K^e} = \int_{\mathbf{V_e}} \mathbf{B^T D B dV} \tag{10}$$

In 2D plane stress with quadrilateral elements and plate thickness t, the equation can be simplified to:

$$\mathbf{K^e} = \mathbf{t}\int_{\mathbf{-1}}^{\mathbf{1}}\int_{\mathbf{-1}}^{\mathbf{1}} \mathbf{B^T D B |J| d}\eta\mathbf{d}\xi \tag{11}$$

Where $|J|$ is the determinant of the Jacobian of the mapping between local and global coordinates. Similar equations can be derived for other element types. For more information refer to [1] and [3].

3 A MATLAB implementation

The MATLAB program described here illustrates some implementation details for the finite element formulation described in the last section. This program was checked first by checking the displacements in a simple bar in tension problem, then with ANSYS by solving some 2D linear plane stress problems, with identical meshes. The program was used to compute the stress field generated by two adjacent holes in an infinite plate.

3.1 Reason for language choice

Since MATLAB is a program specifically designed to work with matrices (the name stands for MATrix LABoratory), it is a good choice for FEA work. It is also easier to learn than other lower level languages like C/C++ and FORTRAN. As long as the global stiffness matrix is not too large, the difference in execution speed between MATLAB and lower level languages is not critical. MATLAB is slower because it is an interpreted language that uses recompiled functions. To write a program that is faster therefore means that the code should be as concise as possible and if and for loops should be avoided (since they are interpreted for every loop). This is called vectorization; using matrix operations instead of for loops and if statements to speed up MATLAB code. Particular examples of this approach are described in the following sections. This implementation is based on [4].

3.2 FEA functions and classes

The MATLAB program was divided into three main functions getmesh(), fep() and fepp(). The first function is used to read node and connectivity files. The second function implements the actual processing and fepp() calculates the nodal stresses (the extra p stands for post-processing; usually referred to as any computation done after solving for the displacements using the global stiffness matrix).

3.3 FEA algorithms

The main processing algorithm is given as follows:

```
%GENERATE K MATRIX
%Get quadrature points in 2D (first number is order of entries in k^e)
[qW,qP] = quadrature(gaussDeg,gaussType,2);
%Generate ke matrices and put directly into K matrix
for e = 1:nElements
    %k quadrature loop
    for q = 1:size(qW,1)
        %Calculate derivatives of shape functions
        [N,dNdxi] = lagrange_basis(elemType,qP(q,:));
        elemNodes = p(t(e,:)',:);
        J         = (elemNodes'*dNdxi)';
        invJt   = inv(J');
        dNdx  = dNdxi*invJt;
        %Make B matrix for element
        B = zeros(3,2*nPerElem);
        B(1,1:2:(2*nPerElem-1)) = dNdx(:,1)';
        B(2,2:2:2*nPerElem    )   = dNdx(:,2)';
        B(3,1:2:(2*nPerElem-1)) = dNdx(:,2)';
        B(3,2:2:2*nPerElem    )   = dNdx(:,1)';
        %Generate K matrix
        %Global location vector
        gL(2*(1:nPerElem)-1) = 2.*t(e,:)-1;
        gL(2*(1:nPerElem))     = 2.*t(e,:);
        K(gL,gL) = K(gL,gL) + B'*D*B*qW(q)*det(J);
    end
end
```

The main processing algorithm uses two for loops to generate the element stiffness matrix for each element and for each quadrature point. To save computation time and space, the global stiffness matrix is assembled directly using a 'global location vector' that for 2D problems and the nodal arrangement used in $\mathbf{U^g}$ is:

```
gL(2*(1:nPerElem)-1) = 2.*t(e,:)-1
gL(2*(1:nPerElem))     = 2.*t(e,:)
```

Instead of having another for loop inside two for loops, the shape function derivatives in terms of intrinsic coordinates where calculated using the following expression:

$$\frac{\partial \mathbf{N}(\xi,\eta)}{\partial \mathbf{x},\mathbf{y}} = \frac{\partial \mathbf{N}(\xi,\eta)}{\partial \xi,\eta}(\mathbf{J}^{\mathbf{T}})^{-\mathbf{1}} \tag{12}$$

Instead of the usual expression that only produces shape function derivatives in terms of intrinsic coordinates for one node:

$$\frac{\partial \mathbf{N_i}(\xi,\eta)}{\partial \mathbf{x},\mathbf{y}} = \mathbf{J}^{-\mathbf{1}}\frac{\partial \mathbf{N_i}(\xi,\eta)}{\partial \xi,\eta} \tag{13}$$

This is an example of vectorization. This is done in [4] but in the wrong way since the actual Jacobian is used instead of its transpose. The shape functions and integration points were obtained from MATLAB functions given by [4]. The shape functions for the 8 noded quadrilateral were added for this implementation. Their derivatives with respect to the local coordinates were calculated using MATLAB's symbolic toolbox. The system is solved using the deceptively simple line of code:

```
U = K\F;
```

As mentioned before, MATLAB can and does use various solution algorithms based on matrix properties such as bandwidth, symmetry and sparseness that allow for a smaller computation time. For more information consult [5].

The equivalent nodal loads from the applied pressure were calculated using the equivalent work principle to obtain consistent loading:

$$\mathbf{q} = \int_{\mathbf{L}} \mathbf{N}\mathbf{P}(\mathbf{x})\mathbf{dx} \tag{14}$$

This was done in MATLAB as follows:

```
%GENERATE EXTERNAL FORCE VECTORS
%Sort force nodes based on x value
[fbNxs,indices] = sort(p(forcebNodes,1));
%Calculate lengths between nodes
numfNs = length(forcebNodes);
ls     = diff(fbNxs);
nodalForces = zeros(1,numfNs);
%Choose between linear or quadratic elements
switch elemType
    case {'T3','Q4'}
        %Calculate loads in nodes (consistent loading) adding forces
        %from sides (linear case)
        n = 2:(numfNs-1);
        nodalForces(1)      = ls(1)/2;
        nodalForces(n)      = (ls(n-1) + ls(n))/2;
        nodalForces(numfNs) = ls(numfNs-1)/2;
    case {'T6','Q8'}
        %Calculate loads in nodes (consistent loading) adding forces
        %from sides (quadratic case)
        n = 3:2:(numfNs-2);
        nodalForces(1)      = qholder(1, 1, ls);
        nodalForces(2)      = qholder(2, 1, ls);
        nodalForces(n)      = qholder(3, n-2, ls) + qholder(1, n, ls);
        nodalForces(n+1)    = qholder(2, n, ls);
        nodalForces(numfNs) = qholder(3, numfNs-2, ls);
end
F(forcebNodes(indices)*2) = nodalForces.*sigma;
```

The use of the MATLAB functions sort() and diff() prevent the use of a for loop. This is another example of vectorization. The qholder function was written to make the code more readable. For example, qholder can contain the expression used to obtain consistent loading for quadratic elements:

```
function result = qholder(type, n, ls)
%Function helps calculate distributed loads (/const pressure)
%for quadratic elements
switch type
    case 1
    result = (ls(n)+ls(n+1))./ls(n).*(ls(n)./3 - ls(n+1)./6);
    case 2
    result =
    (ls(n)+ls(n+1)).*(ls(n)+ls(n+1)).*(ls(n)+ls(n+1))./6./ls(n)./ls(n+1);
    case 3
    result = (ls(n)+ls(n+1))./ls(n+1).*(ls(n+1)./3 - ls(n)./6);
end
```

The fixed boundary conditions were applied by changing the matrix size as follows:

```
%APPLY BOUNDARY CONDITIONS
%Zero rows and columns on y component of fixed nodes in K matrix
%Zero leftmost node in x component to prevent free body motion in x axis
%Not needed for F vector since corresponding positions are already zero
[dummy,index] = min(p(fixedbNodes,1));
K(fixedbNodes(index)*2-1,:) = 0; K(:,fixedbNodes(index)*2-1) = 0;
K(fixedbNodes*2,:)             = 0; K(:,fixedbNodes*2)          = 0;
%Put ones in diagonals corresponding to fixed components of nodes
K(fixedbNodes(index)*2-1,fixedbNodes(index)*2-1) = 1;
K(fixedbNodes*2,fixedbNodes*2) = speye(length(fixedbNodes));
```

If one wants to check the analytic solution of a hole in an infinite plate with an FEA solution of the same problem, the ratio between the width of the plate and the hole diameter has to be high enough to approximate an infinite plate. If this is the case, the stress on the boundary of the plate will approach the stress on the boundary of a plate without a hole. instead of checking for the stress at the boundaries, the related displacements were used. This is because the error of the calculated stress is usually higher than the calculated displacement. This was implemented as follows:

```
%GET EDGE LOCATION VECTORS
%Tolerance used in boundary selection and y coordinate of boundaries
btol = 1e-4; edge1 = -1; edge2 = 1;
%Left edge
leftNodes   = find(abs(p(:,1)-edge1) < btol);
%Right edge
rightNodes  = find(abs(p(:,1)-edge2) < btol);
%Top edge
topNodes    = find(abs(p(:,2)-edge1) < btol);
%Bottom edge
bottomNodes = find(abs(p(:,2)-edge2) < btol);

%Calculate max displacements
maxdy = sigma*2/E;
maxdx = -nu*maxdy;

%Compare with actual values
[leftNs,index]   =
sort(p(leftNodes,2));
maxLeftDiff      =
max(abs( (U(leftNodes(index)*2)-(leftNs+1)./2*maxdy)./maxdy ));

(Same for all other sides)

maxDiff = max([maxLeftDiff; maxRightDiff; maxTopDiff; maxBottomDiff]);
```

4 Meshing in MATLAB

4.1 Considerations in choosing a meshing algorithm

The quality of the mesh has to be adequate; if its not, among other problems, the calculated Jacobian at a given quadrature point will produce values approaching either machine epsilon or ceiling, giving the wrong displacements. The quality of a mesh is usually measured by the skewness ratio of the element, the internal angles of the element and the value of the determinant of the Jacobian. ANSYS can produce triangular and quadrilateral meshes in a short amount of time. It also provides warnings if the elements are not well shaped [6]. A meshing algorithm implemented in MATLAB called DistMesh was found to produce better quality elements than ANSYS.

4.2 DistMesh

The meshing algorithm called DistMesh is implemented as a MATLAB function. For more information refer to [7]. The algorithm uses the analogy of a triangle mesh as a truss structure.

The algorithm starts placing nodes randomly with and average distance between nodes of l_0, or weighting the initially uniform probability distribution with the inverse square of the element size function $h(x, y)$ in a bounding box with the shape to be meshed. The element size function is used to change the distance between elements as a function of their position in the bounding box or in the area to be meshed. Then the points used are the ones inside the area to be meshed. This is determined by the distance function $d(x, y)$ of the area to be meshed. It is a function that provides the shortest distance from a point to the boundary of the area to be meshed. The distance is positive if the point is inside the area and negative otherwise. This equation can also be used as part of the element size function. Then the nodes are connected to form a truss structure using Delaunay triangulation. Afterwards the corresponding truss problem is solved for equilibrium: the sum of the forces in each truss is minimized. The force function used is a linear spring function with spring constant k equal to 1:

$$f(x, y) \tag{15}$$

In that way the truss structure spreads out covering the area to be meshed. In order for the nodes to spread far enough, the actual lo inputted in the algorithm's MATLAB function is multiplied by a scaling factor Fscale. A satisfactory value for this factor was found by trial and error for 2D areas to be 1.2. If a node goes outside the area to be meshed, the node is placed at the boundary of the area. This is done also using the distance function to determine a normal force. If the truss connected to the node is not perpendicular to the tangent of the area boundary at that point, the net effect will be that a node that moves past the area will start to move tangent to the area's boundary.

The equation

$$F(p) = 0 \tag{16}$$

is solved with the steepest (or gradient) descent method:

$$F(p) = 0 \tag{17}$$

Which is equal to using an artificial time dependence and a forward Euler method as used in the paper [7]:

$$p_1 = p_n + tF(p) \tag{18}$$

For more information about the steepest descent method and the forward Euler method consult [8]. This iteration is carried out until all trusses expand more than ttol scaled with l_0. Then the Delaunay algorithm is applied again and then the forward Euler iteration. This is repeated until all movements are smaller than dptol scaled with l_0.

Solving the equivalent truss problem for equilibrium and using the Delaunay algorithm produces very uniform elements. It is actually superior in the shapes tested compared to most other common algorithms [7]. Some disadvantages of this method are its execution speed and the fact that a global minimum element quality value cannot be guaranteed. However the method could produce high quality meshes faster if another method is used to get an initial mesh and then DistMesh is used only for mesh refinement. The method can also be used to produce high quality quadrilateral elements as done in the LehrFEM MATLAB function library, using this simple algorithm: generate a triangle mesh using DistMesh, out of all possible quadrilaterals obtained by merging two triangles find the one with the highest quality, merge the triangles corresponding to that quadrilateral and repeat from step two until all triangles are merged into quadrilaterals. Then split all quadrilaterals and any remaining triangle into 4 and 2 quadrilaterals respectively. Unfortunately, the whole LehrFEM library including quadrilateral mesh generation is not available online. To obtain a copy of the complete library contact Dr. Hiptmair at: hiptmair@sam.math.ethz.ch.

References

[1] S. A. Meguid. The finite element method: theory and practise. 2008.

[2] Ansel C. Ugural and Saul K. Fenster. *Advanced Strength and Applied Elasticity.* Prentice Hall, 4th edition, 2003.

[3] R.L. Taylor O. C. Zienkiewicz and J.Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals.* Elsevier, 6th edition, 2005.

[4] Jack Chessa. Programming the finite element method with matlab. 2002.

[5] Timothy A. Davis. *Direct Methods for Sparce Linear Systems.* SIAM, 1st edition, 2006.

[6] Peter Kohnke. Ansys, inc. theory reference. 2004.

[7] Per O. Persson and Gilbert Strang. A simple mesh generator in matlab. *SIAM Review*, 46(2):329–345, 2004.

[8] J.Douglas Faires and Richard Burden. *Numerical Methods.* Brooks/Cole, 2nd edition, 1998.