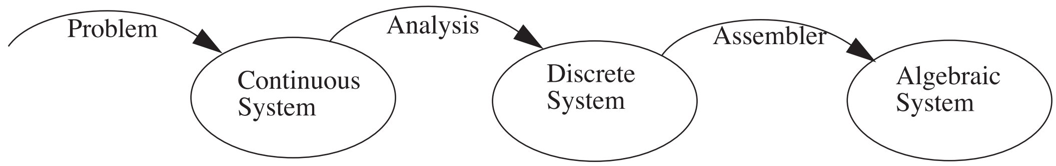
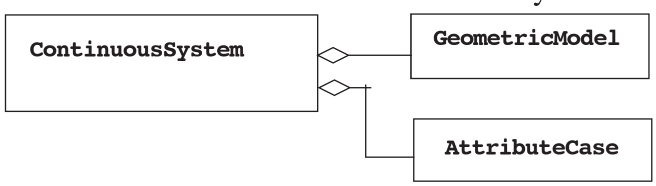
Analysis Procedure

This set of notes takes a geometry-based, object-oriented view of the implementation of finite elements which has the flavor similar to that used in Trellis (which is even more general than what is provided here). In your class assignment, you may wish to take advantage of the fact that your elements are only first or second order defined by interpolating polynomials to simply implementation of some of the steps.

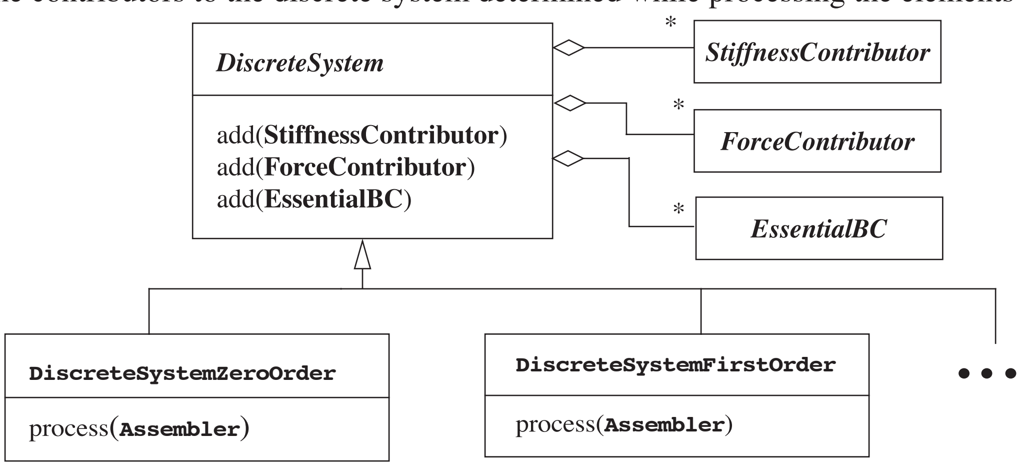
Viewing the analysis as a three step transformation process



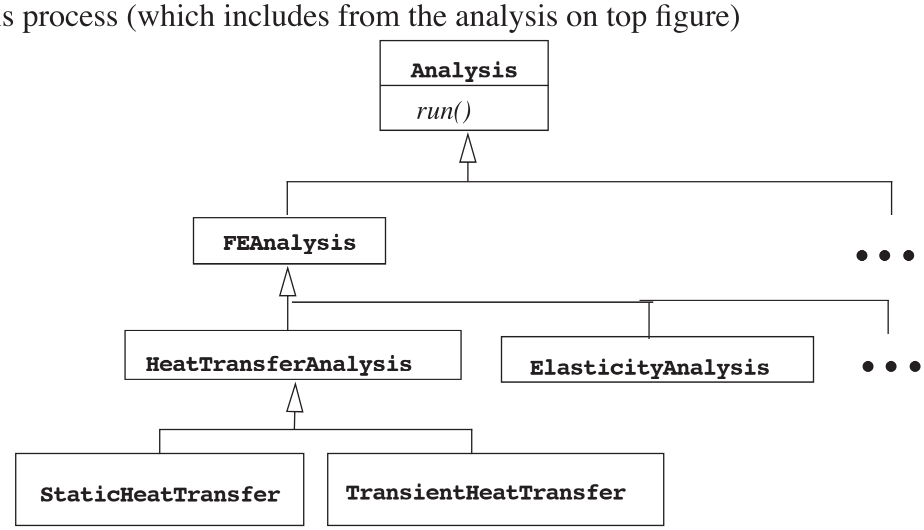
The problem definition comes from the continuous system



The contributors to the discrete system determined while processing the elements



The analysis process (which includes from the analysis on top figure)



The FEAnalysis given assumes a 2-D mesh with mesh faces as stiffness contributors

FEAnalysis
run() *setup() // determines which mesh entities are contributors* *solve() // processes the contributors to set-up, evaluate and solve* *recover() // recovers secondary variables based on the solved for primary variables* *makeStiffContrib(MeshFace face) : StiffnessContributor // face “element” stiffness* *makeForceContrib(MeshFace face) : ForceContributor // face body force* *makeForceContrib(MeshEdge edge) : ForceContributor // edge forces* *makeConstraint(MeshEdge edge) : Constraint* *makeConstraint(MeshVertex vertex) : Constraint* member functions
DiscreteSystem DS; *Mesh theMesh;*

Procedure to perform the analysis

```
FEAnalysis::run
{
  setup(); // determine the contributors
  solve(); // process the contributors to determine the system,
           // perform integrations over contributors, assemble and solve system
  recover(); // recovers any secondary variables
}
```

The class ElasticityAnalysis is derived from FEAnalysis and implements the specialized functionality of an analysis for the case of linear elasticity.

ElasticityAnalysis
makeStiffContrib(MeshFace face) : StiffnessContributor *makeForceContrib(MeshFace face) : ForceContributor* *makeForceContrib(MeshEdge edge) : ForceContributor* *makeConstraint(MeshEdge edge) : Constraint* *makeConstraint(MeshVertex vertex) : Constraint* *recover()*

Better to loop over geometric model entities

For any analysis we need to loop over the mesh entities to determine which are stiffness contributors, force contributors and constraints. These are then added to the discrete system. The procedure given assumes (i) its a 2-D problem, (ii) the mesh faces are the only ones that are stiffness contributors (that is elements in the classic terms), (iii) faces may have body loads, (iv) the mesh edges may be either constrained or “loaded” and the mesh vertices can only have constraints.

It is important to note that setup simply initializes the system contributors by associating the information fundamental to their construction to them. This process is required when supporting a highly flexible set of contributor specifications in terms of the analysis attributes associated with the entities, the discretized fields associated with them, the shape functions used to define the fields over them, and the mappings used to define the geometric jacobian in the integration processes. In your assignment where you are considering only a limited set of specific element shape functions, isoparametric mappings only, and a simpler set of attributes you can simplify the details of the setup process.

The setup and make* functions given here assume only knowledge of the classification of the mesh entity to the geometric model. If there where a list of mesh entities classified on each geometric model entity available, this process could be made simpler by traversing over the geometric model entities performing the appropriate make* operations based on the analysis attributes applied to the model entity.

```
void FEAnalysis::setup()
{
  int i;
  StiffnessContrib sc;
  ForceContrib fc;
  Constraint c;
  for(i=0; i < theMesh->nFace(); i++){ // process all the mesh faces
      MeshFace f= theMesh->face(i); // get face object
      sc = makeStiffContrib(f); // set-up type of stiffness contributor for the face
      DS->add(sc); // all faces contribute to the stiffness - add it to that list
      fc = makeForceContrib(f); // set-up force contributor for the face
      if(fc) // if the face does have a "body" load on it, add to that list
          DS->add(fc);
  }
  for(i=0; i < theMesh->nEdge(); i++){
      MeshEdge e= theMesh->edge(i);
      fc = makeForceContrib(e); // set-up force contributor for the edge
      if(fc) // if the edge is loaded, add to the list of force contributors
          DS->add(fc);
      c = makeContraint(e); // set-up constraint contributor for the edge
      if(c)
          DS->add(c); // if the edge is constraint, add to the list of constraints
  }
  for(i=0; i < theMesh->nVertex(); i++){
      MeshVertex v= theMesh->vertex(i);
      c = makeContraint(v); // set-up constraint contributor for the vertex
      if(c)
          DS->add(c); // if the vertex is constraint, add to the list of constraints
  }
}
```

Note - this could be made more efficient using loops over model entities and reverse classification

go over mesh faces

go over mesh edges

go over mesh vertices

Note that this procedure is the same for all analyses, the only difference is the type of stiffness contributor, force contributor or constraint that is created. These are created by a call to the member functions makeStiffContrib(...), makeForceContributor(...) and makeConstraint(...) which are implemented in the derived classes.

note - the stiffness matrix controibutions are not yet evaluated.

Each of the make* functions in ElasticityAnalysis needs to create the appropriate type of system contributor. For example makeStiffContrib(...) will create the appropriate type of stiffness contributor which may be based on the topology of the face. All the make* functions return information on the shape functions used to define the variation of the solution parameters over the mesh entity and information of the mapping function used to describe its shape.

```
ElasticityAnalysis::makeStiffContrib(MeshFace face)
{
 ShapeFunction sf= // make right kind of shape function
 Mapping mapping= // make right kind of mapping
 return new ElasticitySC(face,mapping, sf);
 }
```

Similarly the makeForce* and makeConstraint(...) functions, must examine the model entity that the mesh entity is classified on and look at the attributes on that model entity to determine the type of system contributor to return.

```
ElasticityAnalysis::makeForceContrib(MeshFace face)
{ // will create a force contributor if the face is loaded
 if (force attribute on face){
     ShapeFunction sf= // make right kind of shape function
     Mapping mapping= // make right kind of mapping
     Attribute attri= // puts information point to the attribute so appropriate values can be calculated
     return ElasticityFC(face,mapping,sf,attri)
 }
}

ElasticityAnalysis::makeForceContrib(MeshEdge edge)
{
 if (edge not classified on model edge) // has to be on the boundary of the domain to have a traction
     return 0; // no force contributor
 if (force attribute on edge){
     ShapeFunction sf= // make right kind of shape function
     Mapping mapping= // make right kind of mapping
     Attribute attri= // puts information point to the attribute so appropriate values can be calculated
     return new ElasticityFC(edge,mapping,sf,attri)
 }
}

ElasticityAnalysis::makeConstraint(MeshEdge edge)
{
 if (edge not classified on model edge) // has to be on the boundary of the domain to be constrained
     return 0; // no constraint
 if (edge constrained){
     ShapeFunction sf= // make right kind of shape function
     Mapping mapping= // make right kind of mapping
     Attribute attri= // puts information point to the attribute so appropriate values can be calculated in the case of
                  // non-zero boundary constraints
     return new DisplacementConstraint(edge,mapping,sf,attri)
 }
}
```

```
ElasticityAnalysis::makeConstraint(MeshVertex vertex)
{
  if (vertex not classified on model edge or model vertex) // has to be on the boundary domain to be constrained
        return 0; // no force constraint
  if (vertex constrained){
        ShapeFunction sf= // make right kind of shape function
        Mapping mapping= // make right kind of mapping
        Attribute attri= // puts information point to the attribute so appropriate values can be calculated in the case of
                        // non-zero boundary constraints
        return new DisplacementConstraint(vertex,mapping,sf,attri)
  }
}
```

At this point the analysis is set up (the DiscreteSystem has been defined in terms of the contributors). We now wish to transform the DiscreteSystem into an AlgebraicSystem (set up the linear algebra). This is implemented in the solve() member function of the analysis class. For example:

```
ElasticityAnalysis::solve()
{
  LinearSystemAssembler assembler; // need to have the appropriate assembly class
  AlgebraicSystem AS(DS,assembler); // this class will contain the correct structure for the global system
  AS.solve();
}
```

The two new classes of LinearSystemAssembler and AlgebraicSystem introduced here need to be described. First we'll look at AlgebraicSystem. This class represents the matrix equation $Kd = f$. It's solve() function assembles the global system and invokes a solver to solve it.

```
AlgebraicSystem::solve()
{
  DS->initializeSystem();
  createGlobalSystem(); // create and zero the system in preparation for the summations of the assembly process
  A-> initialize(K,f) // tells the assembler where the global stiffness matrix and load vectors are
  DS->formSystem(A);
  solveLinearSystem();
}
```

The first step is to get the discrete system to process all of the constraints for the system. This must be done before creating the global stiffness matrix and global force vector. Next, with the constraints applied, the global matrix and vectors are created by calling createGlobalSystem(). It is then necessary to tell the assembler which global matrix and vector it is assembling into. This is done by calling Assembler::initialize(...). Next, DiscreteSystem::formSystem(...) is called passing the initialized assembler. This causes the discrete system to evaluate all the system contributors with the given assembler. Finally the resulting linear system is solved.

AlgebraicSystem

AlgebraicSystem(DiscreteSystem ds, Assembler assem)
solve()
createGlobalSystem()
solveLinearSystem()

DiscreteSystem DS; // gets us the various contributors
Assembler A; // gets us the assembler for our problem
SparseMatrix K; // the stiffness matrix in a proper structure
Vector d; // vector of unknowns to be solved for
Vector f; // force vector(s) - may be multiple RHS cases

DiscreteSystem

add(StiffnessContributor sc)
add(ForceContributor fc)
add(Constraint c)

initializeSystem()
formSystem(Assembler a)

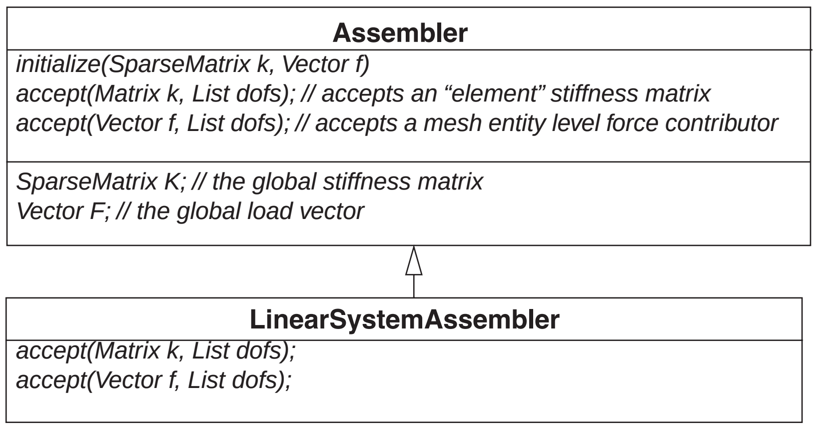
List SCList; // list of stiff. contrib
List FCList; // list of force contrib
List CList; // list of constraints

```
DiscreteSystem::initializeSystem()
{
 for(each constraint)
      CList[i]->apply(); // each essential boundary condition will eliminate possible dof from the global system
                         // in the case of non-zero essential boundary conditions must also maintain the non-zero value
}

DiscreteSystem::formSystem(Assembler assem)
{ // evaluates the force and stiffness contributors, as each one is evaluated it is (in this code) immediately
 // assembled in the to appropriate positions of the global system
 // note that the contributions to the force vector due to non-zero essential bc will also be done at this time using
 // the approiate stiffness terms for the stiffness contributors times the appropriate non-zero esential bdry condition
 for(each force contributor)
      FCList[i]->evaluate(assem); // perfrom the operations required to evaluate each for
 for(each stiffness contributor)
      SCList[i]->evaluate(assem);
}
```

The purpose of the assembler class is to take the contributions of each force and stiffness contributor and assemble it into the global system. Assembler has two member functions called accept(...), the first takes a matrix and a list of degrees of freedom, the second takes a vector and a list of degrees of freedom. The first variation of this function corresponds to assembling a stiffness contributor and the second corresponds to assembling a force contributor.

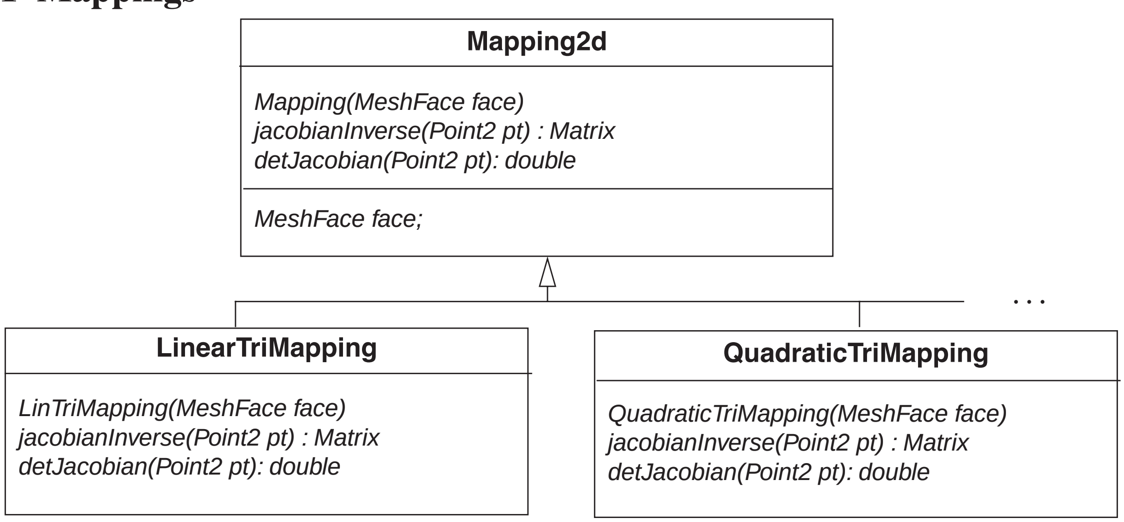
The list of degrees of freedom is used by the assembler to figure out if and where to assemble each term in the matrix into the global matrix. The assembler will be explained later when we have all the needed pieces explained. The assembler shown here is a limited to a simple linear system. In a full implementation there are subclasses to support the assembly process for various semi-discrete systems as seen in time dependent problems where time is discretized by a difference operator.



1. Stiffness Contributors

The stiffness contributors are evaluated by the integration over the domain of the mesh entity associated with the stiffness contributor. As seen in class, one needs the shape functions, the geometric mappings, knowledge of the integrand to be formed and the appropriate integration rule to evaluate the stiffness contributors. These items are discussed here.

1.1 Mappings



The mapping classes represent the transformation from the local coordinate system of the shape functions to the global coordinate system that PDE is written in. There are two important member functions in a mapping: jacobianInverse(Point2 pt) returns the inverse of the jacobian of the mapping at a certain point, detJacobian(Point2 pt) returns the determinate of the jacobian of the mapping at a point. Both of these member functions must be implemented for each specific type of mapping.

1.2 Shape Functions

Shape functions represent the interpolation of degrees of freedom over a mesh entity in a particular local coordinate system. These are setup back in MakeStiffContrib.

When the shape function is constructed the constructor (LinearTriSF for example) will also initialize any DOF objects not already constructed. Care must be taken during this process to be sure that the DOF object was not already constructed by a previously processed stiffness contributor. This can happen when the shape function is associated with something on the boundary of the element. With our current assumptions of up to 2 dof per node, you may want to use a simpler procedure that simply sets up the two DOF objects for each node. Note that this is not as general as the what is implied here.

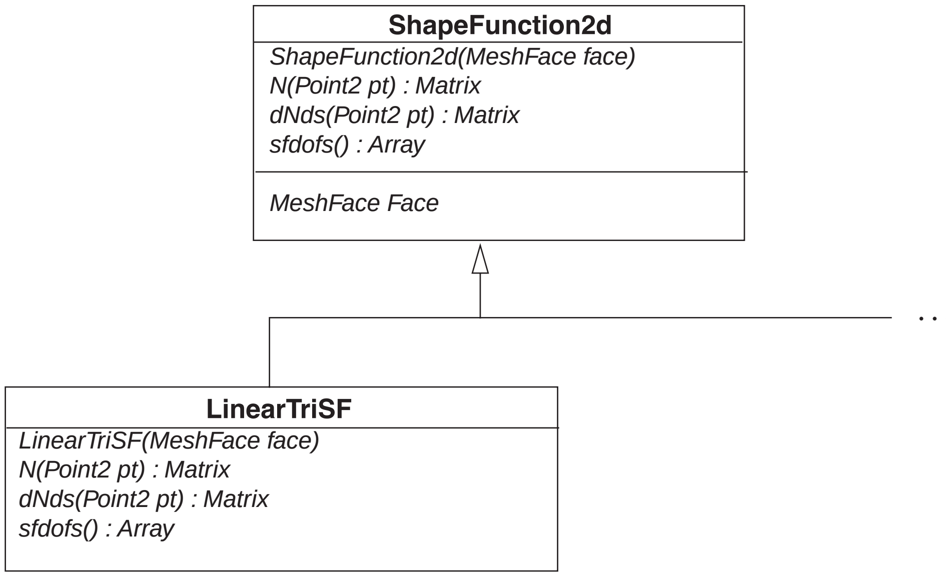
ShapeFunction2d::sfdofs() - returns an array of the degrees of freedom used by the shape function in the order that they are returned by the functions N() and dNds(). Each row in the array may contain multiple possible dof. In the case we are considering where there is one dof per each of the two components, the array sfdofs will store the global dof objects (DOF) for each of them. By the time sfdofs is used to construct the map of the local to global dof, the DOF objects will have been processes to contain the approptiate information. The degrees of freedom are returned in an array where the rows correspond to the ordering of the shape functions returned by N and dNds and the columns correspond to the degrees of freedom at each node.

ShapeFunction2d::N(Point2 pt) - returns the shape functions evaluated at the point pt. This is returned as a matrix:

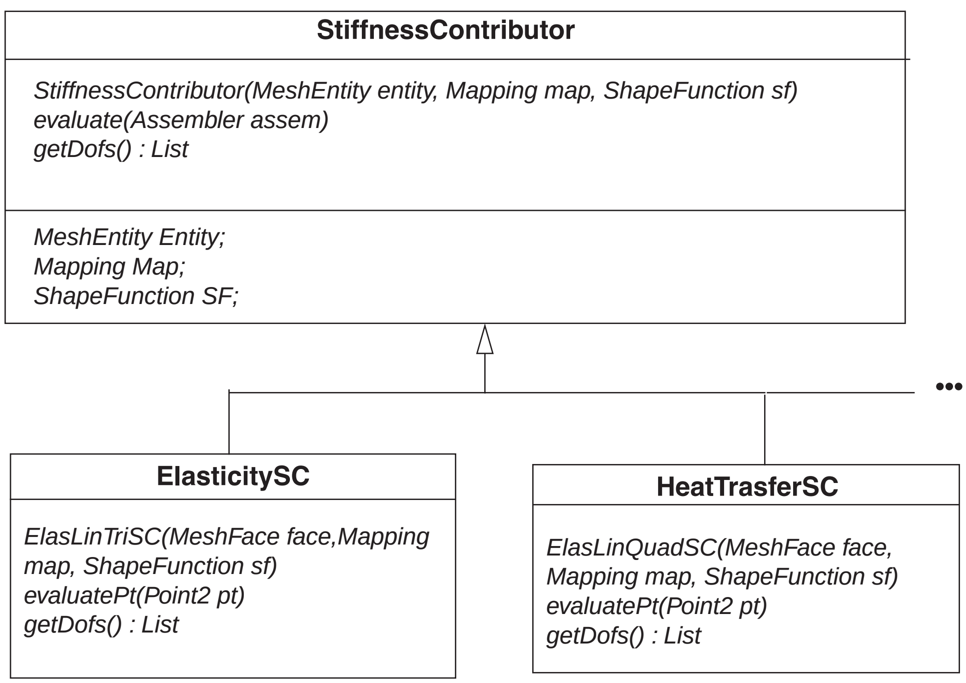
$$\begin{bmatrix} N_1(pt) & N_2(pt) & N_3(pt) & \dots & N_n(pt) \end{bmatrix}$$

ShapeFunction2d::dNds(Point2 pt) - returns the first derivative of the shape functions (in the local coordinate system) evaluated at the point pt. This is returned as a matrix:

$$\begin{bmatrix} N_{1,r}(pt) & N_{2,r}(pt) & N_{3,r}(pt) & \dots & N_{n,r}(pt) \\ N_{1,s}(pt) & N_{2,s}(pt) & N_{2,s}(pt) & \dots & N_{n,s}(pt) \end{bmatrix}$$



1.3 Evaluation of the Stiffness Contributors



The stiffness contributors use the shape functions and the mappings to calculate the coupling between degrees of freedom.

The getDofs() function returns the degrees of freedom for the stiffness contributor in the order corresponding to that of the calculated local stiffness matrix. That is, an entry in the local stiffness matrix k_{ij} couples the i^{th} and j^{th} degrees of freedom as returned in this list.

Assuming all integrations are done using some type of quadrature, the evaluate(...) function in StiffnessContributor can be written as:

```
StiffnessContributor::evaluate(Assembler assem)
{
 Matrix k; // assume its initialized to zero
 List dofs = getDofs();
 for(i=0; i < numIntPts){ // looping over the number of integration points
     Point2 pt = // i'th integration point
     double weight = // i'th integration weight
     k += Map->detJacobian(pt)*evaluatePt(pt)*weight // add the proper contribution to the entity stiffness matrix
                                   // note that evaluatePt(pt) represents the evaluation of transpose(B)DB
                                   // at the point
 }
 assem->accept(k,dofs); // completed entity stiffness matrix is given to the assembler to add into the global system
}
```

This function will work with the calculation of all stiffness contributors. This function calls the function StiffnessContributor::evaluatePt(...) to evaluate the stiffness contributor at a specific point within the domain of the mesh entity. The specifics of the mathematics within each stiffness contributor are implemented within evaluatePt(...) for each derived class of StiffnessContributor.

For example in the case of Elasticity, evaluatePt would be written something like:

```
Matrix ElasticitySC::evaluatePt(Point2 pt)
{
 Matrix dNdx = Map->jacobianInverse()*SF->dNds(pt);
 Matrix B = // symmetric gradient
 Matrix D= // material props
 // do the work to form transpose(B)*D*B at the point
 // you will want to take care in the development of your code to properly structure the formation of this
 return transpose(B)*D*B;
}
```

2. Force Contributors

Do this stuff exactly the same way as the stiffness contributors. As long as we make comparable shape functions when we create the force contributors then everything works fine. If the load is over the “element’s” domain, its the element shape functions. If the load is over the one of the mesh entities bounding the element, the shape functions need to be equivalent to the element shape functions evaluated over the domain of the boundary entity only.

3. Degrees of Freedom

DOF
setStatus(int s) setValue(double v) DOF()
double Value; int Status; int EqNumber;
global int ndof;

Objects of the DOF class represent the potential degrees of freedom in the global system. Each DOF has a value, a status and a global equation number. The status of a DOF can take on three values: Free, Zero, and Fixed. “Free” indicates that the degree of freedom is not constrained in any way, “Zero” indicates that the degree of freedom is constrained to be identically zero, it’s value should also then equal zero, “Fixed” indicates a degree of freedom that has a fixed value that may be other than zero, it’s value should be set to the correct value. The individual objects of the class DOF were actually set-up back in makeStiffContrib when the shape functions over the mesh entities were being set-up. As this was being done the status of each DOF was set to Free and the global variable ndof was being incremented for each DOF object created. Therefore at the end of the process of setting up the stiffness contributors the variable ndof is set to the value of the maximum number of possible degrees of freedom in the problem. As the constraints are processed, the Status of appropriate DOF objects are changed from Free to Zero, or Fixed, and the global variable ndof is decremented by one each time a status is changed from Free.

Each node will have one or more DOF objects to indicate the degrees of freedom that exist at that node. For example, in 2-D elasticity with Lagrangian polynomials each node has two and will look like:

Node
(... other stuff a node has...) getDofs() : List DOF dofs[2];

`node = dof holder`

4. Constraints

The Constraint class is very similar to the StiffnessContributor and ForceContributor classes. It is constructed by giving it a shape function and mapping for the mesh entity that it exists on. In addition the constraint must be given some information on the value of the constraint. For DisplacementConstraint it is necessary to be able to apply the constraint to one or more degrees of freedom (since it is constraining a vector quantity).

In the DisplacementConstraint constructor the attribute information, attri, indicates what is constrained, and, with appropriate interpretation, the shape functions dictate which of the potential dof are actually constrained. For the simple case we are doing we have three basic constraint types which you can describe by an integer as follows:

= 1 - constrain x component only
= 2 - constrain y component only
= 3 - constraint x and y components

Using Geometry based attributes to account for boundary Conditions

For completeness you would likely consider BC. applied to model faces, edges and vertices

If one wants to be careful with respect to mathematical modeling BC. would be applied to only model faces in the case of manifold 3D domains, etc.

In the case of essential BC. - The BC should be inherited by the closure of the face.

In the case of natural BC - The BC should only be applied to the mesh entities of the same order - remember we have to integrate natural B.C. to get dof related contributions.

Consider the case of setting the dof for essential BC.

Two cases:

Based on Classification

Based on Reverse Classification

(b)

Based on classification

Begin
Do {traverse over all M_i^2's in the mesh}
Get the next mesh face $- M_i^2$
If $M_i^2 \sqsubset G_j^2$ for any j then {mesh face is on a model face}

If G_j^2 has an essential B.C. then
{Only process if there is an essential BC}

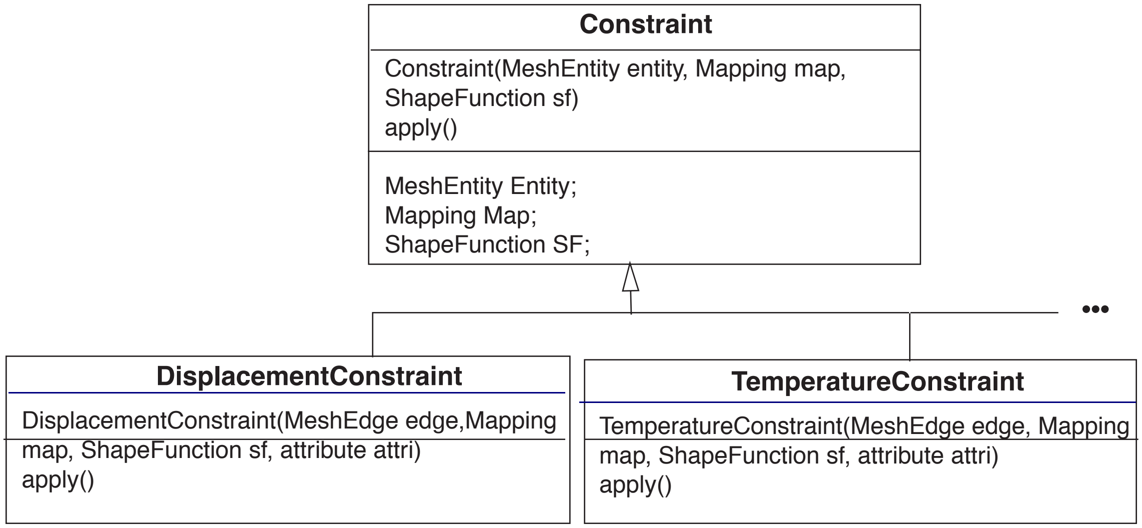
Process dof associated with $\overline{M}_i^2$:

Process the dof associated with M_i^2
Determine $M_k^1\{M_i^2\}$
Loop over the M_k^1 in that set
If dof on that edge not processed yet
Process dof associated with M_k^1
Determ $M_k^0\{M_i^2\}$
Loop over the M_k^0 in that set
If dof on M_k^0 not processed yet
Process dof associated with M_k^0

end If {G_j^2 has an essential BC.}
end If {$M_i^2 \sqsubset G_j^2$}
end Do

Based on Reverse Classification
Do {traverse over the G_j^2 in the model}
If current G_j^2 has an essential BC then
{traverse over the $M_i^2 \sqsubset G_j^2$ from Reverse Classification}
Process the dof associated with $\overline{M}_i^2$ (closure)
end if {G_j^2 has an essential BC
end Do

The constraints must be applied to the appropriate dof as defined by the shape functions on the constrained entity. The values of the non-zero constraints can also be processed then and stored for use.



The apply member function of the constraint must modify the degrees of freedom on the mesh to reflect the application of the constraint.

```
DisplacementConstraint::apply()
{ // this procedure is written based on the assumption of only the three possible simple attributes
  Array sfdofs = SF->sfdofs(); //based on the shape functions for this constraint - get the DOF objects
  if( // x component constrained - that is a 1 or 3 from above){
        for( i = 0; i < dofs.size(); i++){
              if(Values[0] = 0){ // for the case of zero essential bc
                    sfdofs(i,0)->setStatus(Zero) // this sets the status of the correct DOF object
                    sfdofs(i,0)->setValue(0)
              } else { // for the case of non-zero essential bc set the status and value of the correct DOF object
                    sfdofs(i,0)->setStatus(Fixed)
                    sfdofs(i,0)->setValue(value) // value based on the attribute evaluation at location
              }
        }
  }
  if( // y component constrained - that is a 2or 3 from above){ // process same as for x-constraint
        for( i = 0; i < dofs.size(); i++){
              if(Values[1] = 0){ // for the case of zero essential bc
                    sfdofs(i,1)->setStatus(Zero)
                    sfdofs(i,1)->setValue(0)
              } else { // for the case of non-zero essential bc
                    sfdofs(i,1)->setStatus(Fixed)
                    sfdofs(i,1)->setValue(value)
              }
        }
  }
}
```

Set initial status of a DOF object

```
DOF::DOF()
{ // first time a new potential dof is hit set it up and set its status to free - status may change later
  status = Free
```

```
 ndof = ndof + 1
}
```

Reset the Status of a DOF object

Counting the actual dof - originally assumed all possible were dof

```
DOF::setStatus(int s)
{ // resets the status as appropriate
 if (Status = Free and s!=Free) // need to decrement ndof for status changes from free to Zero or Fixed
     ndof = ndof -1
 Status = s
}
```

Set the value for a non-zero essential BC

```
DOF::setValue(double v)
{
 Value = v
}
```

Set-up the information indicating the coupling of dof in the global stiffness matrix

```
AlgebraicSystem::createGlobalSystem()
{
 renumber();// renumber the active degrees of freedom using the same basic methodology as before
 K->setSize(numDof); // just setting the number of dof in the stiffness matrix, not initializing its memory
 for ( // each stiffness contributor, sc, in discrete system) { // want to loop over all the stiffness contributors
     // for the purpose of determining which dof are related to other dof
     // this information is used to determine the maximum column height, or in the case of the row
     // storage given below, the first non-zero column for each row
     List dofs = sc->dofs(); // get the list of DOF objects for the stiffness contributor
     for( int i = 0; i < dofs.size(); i++){ // loop over the number of dof in the dofs
         DOF dof1 = dofs[i]; // get the current dof
         if( dof1->status == Free){ // only see what its coupled to if it is a global dof
             for(int j = 0; j < dofs.size(); j++){// it will be coupled to all the other dof in the contributor
                 DOF dof2 = dofs[j];
                 if(dof2->status = Free){ // again only reflect coupling of global dof
                     K->addNonZeroTerm(dof1->EqNumber, dof2->EqNumber) // updates the
                 }                                   // appropriate structure to reflect the coupling
             }
         }
     }
 }
 K->allocateMemory();
}
```

Example structure - vector of highest column for each row

We will skip the one given here and look at a modified version of what we did before

5. Renumbering the dof

The algorithm given here is the same algorithm as before. The only difference this time is that instead of setting the labels on the nodes, it sets the labels on the DOF.

renumber(mesh)
{ // Reorders equations and elements in a 2-D mesh using the same assumptions as previously
int labeldof = ndof + 1 // value set to the total number of dofs plus 1, this allows auto. reversing
int labelface = nface + 1 // same issues in labeling elements
Node node
MeshEntity entity = getStart( ) // get starting entity to be considered first. Use a $M_i^0 \sqsubset G_j^0$ with min. G_k^1
q $\rightarrow$ enqueue(entity)

```
while (q → size( ) > 0) { // process entities until the queue is empty
 entity = q → dequeue( )
 node = entity → getNode( )
 if (/*dofs on node are unlabeled*/) then { // we are now directly labeling the dof and not the node
     List dofs = node->getDofs()
     for(int idof = 0; idof < dofs->size(); idof++){
         DOF d = dofs[idof]
         if(d->Status = Free){
             labeldof = labeldof -1
             d->EqNumber = labeldof
         }
     }
 }
 // Find unnumbered adjacent mesh entities and label faces. Additions to queue by keying from vertices
 if (entity → dimension( )=0) then { // load adjacent entities by order, label faces & specific edge nodes
     MeshVertex vertex = entity
     for (i = 1 to vertex → numEdges( ) ) { // loop over number of edges using the vertex
         MeshEdge edge = vertex → edge(i)
         for (j = 1 to edge → numFaces( )) do {
             MeshFace face = edge → face(j)
             if (/* face not already labeled*/) then {
                 labelface = labelface -1
                 face → setLabel(labelface)
             }
             if (/* face has node that needs queueing */) then { // queue the face
                 q → enqueue(face)
             }
         }
         othervertex = edge → otherVertex(vertex)
         if (node = edge → getNode( )) then { // if the edge has a node on it
             if (/* othervertex labeled or in queue and edge node not labeled*/ ) then {
                 labeldof = labeldof -1
                 node = edge → getNode( )
                 List dofs = node->getDofs()
                 for(int idof = 0; idof < dofs->size(); idof++){
                     DOF d = dofs[idof]
                     if(d->Status = Free){
                         labeldof = labeldof -1
                         d->EqNumber = labeldof
                     }
                 }
             } else {
                 q → enqueue(edge)
                 list → add(othervertex)
             }
         } else {
             if (/* node at other vertex is not labeled) then {
                 list → add(othervertex)
             }
         }
     } // finished the loop over the edges coming into the current vertex
     q → enqueueList(list) // now queue the other vertices loaded into the list
     emptyList( )
 }
}
```

AdjReorder (this one labels dof - see change bars)

Classes

AdjReorder
getStart( ) : entity // get starting mesh vertex renumber(mesh) // renumbers the nodes and elements
Queue q Mesh mesh

List
add(entity) // adds an entity to a list emptyList( ) // empties a list

Queue
enqueue(item) // enqueues an item into the queue enqueueList(List) // enqueues list into the queue dequeue( ) : item // removes an item from the list size( ) : int // returns the number of entities in the queue

MeshEntity
dimension( ) : int // indicates the dimension of a mesh entity 0-vertex, 1-edge, 2-face, 3-region numEdges( ) : int // indicates the number of edges bounding or coming into an entity numFaces( ) : int // indicates the number of faces bounding or coming into an entity edge(i) : MeshEdge // returns the i th edge bounding or coming into an entity face(j) : MeshFace // returns the j th face bounding or coming into an entity getNode( ) : Node // gets the node on the mesh entity

MeshEdge
otherVertex(vertex) // gets the other vertex for a given edge

Node
setLabel(label) // sets node label to label

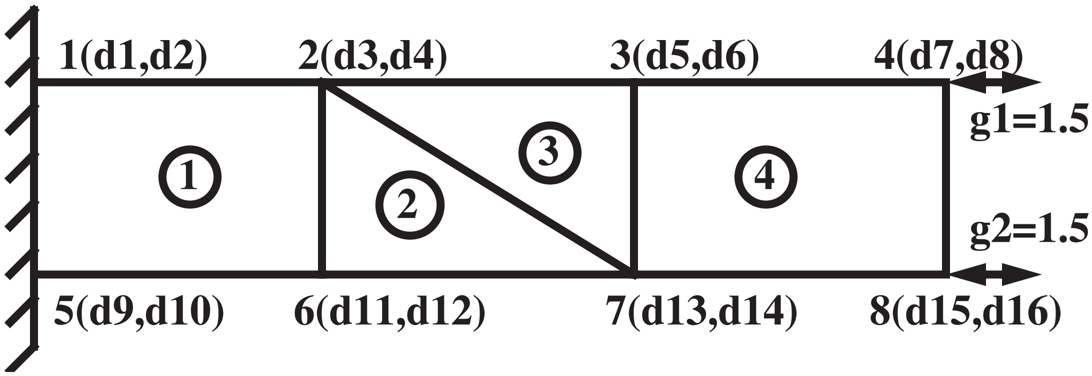
Pseudo Code

renumber(mesh)
{ // Reorders nodes and elements in a 2-D mesh assuming that only the mesh faces are elements
// It is also assumed that all dof associated with an entity are associated with a single node on that entity
int labeldof = ndof + 1 // value set to the total number of dofs plus 1, this allows auto. reversing
int labelface = nface + 1 // same issues in labeling elements
Node node
MeshEntity entity = getStart( ) // get starting entity to be considered first. Use a $M_i^0 \sqsubset G_j^0$ with min. G_k^1
q $\rightarrow$ enqueue(entity)
while (q $\rightarrow$ size( ) > 0) { // process entities until the queue is empty

```
entity = q → dequeue( )
node = entity → getNode( )
if (/*dofs on node are unlabeled*/) then {
    List dofs = node->getDofs()
    for(int idof = 0; idof < dofs->size(); idof++){
        DOF d = dofs[idof]
        if(d->Status = Free){
            labeldof = labeldof -1
            d->EqNumber = labeldof
        }
    }
}
// Want to find any unnumbered adjacent mesh entities and label faces
// All the additions to the queue will be done by looking at adjacencies keying from vertices
if (entity → dimension( )=0) then { // need to load adjacent entities by adjaceny order
    // Also label neighboring mesh faces and specific edge nodes
    MeshVertex vertex = entity
    for (i = 1 to vertex → numEdges( ) ) { // loop over number of edges using the vertex
        MeshEdge edge = vertex → edge(i)
        for (j = 1 to edge → numFaces( )) do {
            MeshFace face = edge → face(j)
            if (/* face not already labeled*/) then {
                labelface = labelface -1
                face → setLabel(labelface)
            }
        }
        if (/* face has node that needs queueing */) then { // queue the face
            q → enqueue(face)
        }
        othervertex = edge → otherVertex(vertex)
        if (node = edge → getNode( )) then { // if the edge has a node on it
            if (/* othervertex labeled or in queue and edge node not labeled*/ ) then {
                labeldof = labeldof -1
                node = edge → getNode( )
                List dofs = node->getDofs()
                for(int idof = 0; idof < dofs->size(); idof++){
                    DOF d = dofs[idof]
                    if(d->Status = Free){
                        labeldof = labeldof -1
                        d->EqNumber = labeldof
                    }
                }
            } else {
                q → enqueue(edge)
```

```
                list → add(othervertex)
            }
        } else {
            if (/* node at other vertex is not labeled) then {
                list → add(othervertex)
            }
        }
    } // finished the loop over the edges coming into the current vertex
    q → enqueueList(list) // now queue the other vertices loaded into the list
    emptyList( )
  }
}
```

6. An Example



This example assumes all shape functions are associated with nodes and there are two possible dof per node. The element (face) labels are shown in circles. The node “labels” are shown next to the nodes. The numbers in the (,) indicates the appropriate DOF object. In this example the faces and equations are labeled to minimize the computational time. The nodes “labels” indicated above are just whatever.

In the sfdof arrays given below it is assumed that the dof for each stiffness contributor (element) are ordered based on traversing the loop around the elements and seeing the order the nodes are traversed. For the specific example: element 1 - 5,6,2,1; element 2 - 6,7,2; element 3 - 7,3,2; element 4 - 7,8,4,3.

The DOF objects after the constraints are processes are:

node its at	DOF object label	equation #	status	value
1	d1	--	Zero	
1	d2	--	Zero	
2	d3	1	Free	
2	d4	2	Free	
3	d5	5	Free	
3	d6	6	Free	
4	d7	--	Fixed	1.5
4	d8	9	Free	
5	d9	--	Zero	
5	d10	--	Zero	
6	d11	3	Free	
6	d12	4	Free	
7	d13	7	Free	
7	d14	8	Free	
8	d15	--	Fixed	1.5
8	d16	10	Free	

The resulting sfdof arrays which contain the labels of the appropriate DOF objects are:

$$
\overset{1}{\begin{bmatrix} d9 & d10 \\ d11 & d12 \\ d3 & d4 \\ d1 & d2 \end{bmatrix}}
\quad
\overset{2}{\begin{bmatrix} d11 & d12 \\ d13 & d14 \\ d3 & d4 \end{bmatrix}}
\quad
\overset{3}{\begin{bmatrix} d13 & d14 \\ d5 & d6 \\ d3 & d4 \end{bmatrix}}
\quad
\overset{4}{\begin{bmatrix} d13 & d14 \\ d15 & d16 \\ d7 & d8 \\ d5 & d6 \end{bmatrix}}
$$

The routine getDof uses the stiffness contributors sfdof arrays to construct the list of pointers to the correct DOF objects as associated with the rows and columns to the stiffness contributors (elements) local stiffness matrix. The specific order depends on the local ordering of equations. For example, one may decide to order the x-component equations followed by the y-component equations. An alternative, used here, is to order all components at the nodes one at a time using the same order for the nodes as used in the construction of sfdof. Doing it this way yields the dofs vectors for the stiffness contributors in our current example:

$$
\overset{1}{\begin{bmatrix} d9 \\ d10 \\ d11 \\ d12 \\ d3 \\ d4 \\ d1 \\ d2 \end{bmatrix}}
\quad
\overset{2}{\begin{bmatrix} d11 \\ d12 \\ d13 \\ d14 \\ d3 \\ d4 \end{bmatrix}}
\quad
\overset{3}{\begin{bmatrix} d13 \\ d14 \\ d5 \\ d6 \\ d3 \\ d4 \end{bmatrix}}
\quad
\overset{4}{\begin{bmatrix} d13 \\ d14 \\ d15 \\ d16 \\ d7 \\ d8 \\ d5 \\ d6 \end{bmatrix}}
$$

Finally it is instructive to see where the various terms in a couple of the element stiffness matrices will go in the global matrix.

For element 1

$$
k^1 = \begin{bmatrix}
\text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
\text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
\text{--} & \text{--} & K_{3,3} & K_{3,4} & K_{3,1} & K_{3,2} & \text{--} & \text{--} \\
\text{--} & \text{--} & K_{4,3} & K_{4,4} & K_{4,1} & K_{4,2} & \text{--} & \text{--} \\
\text{--} & \text{--} & K_{1,3} & K_{1,4} & K_{1,1} & K_{2,1} & \text{--} & \text{--} \\
\text{--} & \text{--} & K_{2,3} & K_{2,4} & K_{2,1} & K_{2,2} & \text{--} & \text{--} \\
\text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
\text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--}
\end{bmatrix}
\begin{bmatrix} d9 \\ d10 \\ d11 \\ d12 \\ d3 \\ d4 \\ d1 \\ d2 \end{bmatrix}
$$

For element 4

$$
k^4 = \begin{bmatrix}
K_{7,7} & K_{7,8} & F_7 & K_{7,10} & F_7 & K_{7,9} & K_{7,5} & K_{7,6} \\
K_{8,7} & K_{8,8} & F_8 & K_{8,10} & F_8 & K_{8,9} & K_{8,5} & K_{8,6} \\
\text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
K_{10,7} & K_{10,8} & F_{10} & K_{10,10} & F_{10} & K_{10,9} & K_{10,5} & K_{10,6} \\
\text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} & \text{--} \\
K_{9,7} & K_{9,8} & F_9 & K_{9,10} & F_9 & K_{9,9} & K_{9,5} & K_{9,6} \\
K_{5,7} & K_{5,8} & F_5 & K_{5,10} & F_5 & K_{5,9} & K_{5,5} & K_{5,6} \\
K_{6,7} & K_{6,8} & F_6 & K_{6,10} & F_6 & K_{6.9} & K_{6,5} & K_{6,6}
\end{bmatrix}
\begin{bmatrix} d13 \\ d14 \\ d15 \\ d16 \\ d7 \\ d8 \\ d5 \\ d6 \end{bmatrix}
$$

Assembly and Solution of Linear Systems Arising from Finite Element Problems

At some point we need to assemble the individual element contributions into the global system and solve it. (There are so-called matrix free methods also – they still have to "solve") Our discussion will assume the process flow for doing that will involve:

1. Performing a preprocessing procedure to determine the dof (accounting for BC), structuring the global system and doing needed initializations.
2. Executing a loop over the appropriate mesh entities to:
    a. Determine the contributions to the global system for those mesh entities
    b. Assemble the individual terms from the element contributions into the global system
3. Solve the global system

Note there can be substantial variations in the process. For example "frontal solvers" begin solving the system as enough is assembled. "Matrix free" method do not quite do the matrix assembly as we will define it.

The most time consuming portion of a FE analysis is the assembly and solution of the global linear system.

Items the will influence your selection of the method to be used to solve the global linear systems include:

- The number of unknowns in the system, particularly when the systems are large enough that we need to use substantial levels of parallelism
  - Direct solvers have a computational growth rate higher that optimal iterative solvers.
  - Direct solvers do not scale as well on high process counts.
- Conditions number of the system – cost of iterative solvers (and even their ability to converge to the solution) is a function of the numerical conditioning of the system.
- The sparseness of the global matrix
- If the system is symmetric.
- In the case of linear problems the number of right hand sides (RHS) – direct solvers can solve multiple RHS with one factorization of the system (the expensive part) and a back substitution for each RHS. Iterative solvers have to iterate for each RHS.
- Hardware available - # processors, type of processors/accelerators, communication hierarchy.

For now we will assume that our systems are small enough that we want to use a direct solver. The global system for finite element problems is quite sparse – thus we want to account for this. If the matrix is full the cost of a direct solve is on the order of n^3, where n is the number of equations while in the case of sparse finite element matrices the cost can be reduced to n^β, with $\beta \geq 1.5$.

Direct Solution of Symmetric Positive Definite Systems

$\underset{\sim}{A}\,\underset{\sim}{x} = \underset{\sim}{b} \qquad [A]\{x\} = \{b\}$

Matrix $[A]_{n\times n}$ is positive definite

$\lfloor v \rfloor [A] \{v\} > 0 \quad \forall \{v\} \neq \emptyset$

Will go through a varient of Gauss elimination

The steps we will follow are

1. Factor the matrix
2. Forward reduction
3. Diagonal Scaling
4. Back substitution

Following Chapt. 11 of Hughes' text

A positive definite symmetric matrix can be written as

$A = U^T D U$ where

U is upper triangular and D is a diagonal matrix with all positive terms

$$U^T = \begin{bmatrix} 1 & & & & & \\ U_{2,1} & 1 & & & 0 & \\ U_{3,1} & U_{3,2} & 1 & & & \\ & & & \ddots & & \\ U_{n-1,1} & U_{n-1,2} & \cdots & & 1 & \\ U_{n,1} & U_{n,2} & \cdots & & U_{n,n-1} & 1 \end{bmatrix} \qquad D = \begin{bmatrix} D_{11} & & & & & \\ & D_{22} & & & 0 & \\ & & D_{33} & & & \\ & 0 & & \ddots & & \\ & & & & D_{n-1,n-1} & \\ & & & & & D_{n,n} \end{bmatrix}$$

$$Ax = b \Rightarrow U^T D U x = b$$

Forward reduction

define $z = DUx$ —— (a)

Then we have

$$U^T z = b$$

$$\begin{bmatrix} 1 & & & & \\ u_{21} & 1 & & & \\ u_{31} & u_{3,2} & 1 & & \\ \vdots & & & \ddots & \\ u_{n,1} & \cdots & \cdots & u_{n,n-1} & 1 \end{bmatrix} \begin{Bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{Bmatrix}$$

obviously we can solve this

$$z_1 = b_1$$

$$z_2 = b_2 - u_{21} z_1 = b_2 - u_{21} b_1$$

$$z_3 = b_3 - u_{31} z_1 - u_{32} z_2$$

$$\vdots$$

$$z_i = b_i - \sum_{j=1}^{i-1} u_{ij} z_j \qquad i = 2(1)n$$

Diagonal Scaling

From eq (a) we have

$$DUx = z$$

Set $Ux = y$ —— (b)

$$Dy = z$$

$$y_i = \frac{z_i}{D_{ii}} \qquad i = 1(1)n$$

Back Substitution

from eq (b) we have

$Ux = y$

$$\begin{bmatrix} 1 & u_{12} & u_{13} & \cdots & u_{1n} \\ & 1 & u_{23} & \cdots & u_{2n} \\ & & \ddots & & \vdots \\ & & 1 & u_{n-2,n-1} & u_{n-2,n} \\ & & & 1 & u_{n-1,n} \\ & & & & 1 \end{bmatrix}$$

start from the bottom

$x_n = y_n$

$x_{n-1} = y_{n-1} - u_{n-1,n}\, x_n$

$\vdots$

$$x_i = y_i - \sum_{j=n}^{i+1\,(\text{by } -1)} u_{ij} y_j$$

$i = n-1\,(-1)\,1$

This all looks easy -

The work - computation that is in factoring A into $U^T D U$

Index version of Crout factorization

(6)

The index version of the Crout factorization is expressed as Factorization of Stiffness matrix

By definition
$$A_{ij} = \underbrace{\sum_{k=1}^{j} U_{ki} D_{kk} U_{kj}}_{} \qquad (1 \le i \le j)$$

This all that is left when accounting for $D_{ij} = 0$ $i \neq j$ and recall we have $[U]^T[D][U]$ still need to use: $U_{ii} = 1$ and $U_{ij} = 0$ $i > j$

where note $A_{ij} = A_{ji}$

$A_{ji} = \sum_{k=1}^{i} U_{kj} D_{kk} U_{ki}$

$$U_{ii} = 1,$$
$$U_{ij} = 0, \qquad \text{for } i > j$$

Useful formulas for programming may be derived by expanding the above for $j = 1, 2, \ldots,$ as follows:

$$A_{11} = U_{11} D_{11} U_{11} = D_{11}$$ For $j=1$

$$A_{12} = \cancelto{1}{U_{11}} D_{11} U_{12} + \cancelto{0}{U_{21}} D_{22} U_{22}$$ For $i=1$, $j=2$
$$= D_{11} U_{12} = A_{21}$$
$$A_{22} = U_{12} D_{11} U_{12} + \cancelto{1}{U_{22}} D_{22} \cancelto{1}{U_{22}}$$ For $i=2$, $j=2$
$$= U_{12} D_{11} U_{12} + D_{22}$$

$$A_{13} = \cancelto{1}{U_{11}} D_{11} U_{13} + \cancelto{0}{U_{21}} D_{22} U_{23} + \cancelto{0}{U_{31}} D_{33} U_{33}$$ For $i=1$, $j=3$
$$= D_{11} U_{13} = A_{31}$$
$$A_{23} = U_{12} D_{11} U_{13} + \cancelto{1}{U_{22}} D_{22} U_{23} + \cancelto{0}{U_{32}} D_{33} U_{33}$$ For $i=2$, $j=3$
$$= U_{12} D_{11} U_{13} + D_{22} U_{23}$$
$$A_{33} = U_{13} D_{11} U_{13} + U_{23} D_{22} U_{23} + \cancelto{1}{U_{33}} D_{33} \cancelto{1}{U_{33}}$$ For $i=3$, $j=3$
$$= U_{13} D_{11} U_{13} + U_{23} D_{22} U_{23} + D_{33}$$

$$A_{14} = \cancelto{1}{U_{11}} D_{11} U_{14} + \cancelto{0}{U_{21}} D_{22} U_{24} + \cancelto{0}{U_{31}} D_{33} U_{34} + \cancelto{0}{U_{41}} D_{44} U_{44}$$ $i=1$, $j=4$
$$= D_{11} U_{14}$$
$$A_{24} = U_{12} D_{11} U_{14} + \cancelto{1}{U_{22}} D_{22} U_{24} + \cancelto{0}{U_{32}} D_{33} U_{34} + \cancelto{0}{U_{42}} D_{44} U_{44}$$ $i=2$, $j=4$
$$= U_{12} D_{11} U_{14} + D_{22} U_{24}$$
$$A_{34} = U_{13} D_{11} U_{14} + U_{23} D_{22} U_{24} + \cancelto{1}{U_{33}} D_{33} U_{34} + \cancelto{0}{U_{43}} D_{44} U_{44}$$ $i=3$, $j=4$
$$= U_{13} D_{11} U_{14} + U_{23} D_{22} U_{24} + D_{33} U_{34}$$
$$A_{44} = U_{14} D_{11} U_{14} + U_{24} D_{22} U_{24} + U_{34} D_{33} U_{34} + \cancelto{1}{U_{44}} D_{44} \cancelto{1}{U_{44}}$$ $i=4$, $j=4$
$$= U_{14} D_{11} U_{14} + U_{24} D_{22} U_{24} + U_{34} D_{33} U_{34} + D_{44}$$

And so on. These formulas may be solved for the U_{ij}'s and D_{jj}'s:

From $j=1$

$$D_{11} = A_{11}$$

From $j=2$

$$U_{12} = \frac{A_{12}}{D_{11}}$$

$$D_{22} = A_{22} - D_{11}U_{12}^2$$

From $j=3$

$$U_{13} = \frac{A_{13}}{D_{11}}$$

$$U_{23} = \frac{A_{23} - U_{12}D_{11}U_{13}}{D_{22}}$$

$$D_{33} = A_{33} - D_{11}U_{13}^2 - D_{22}U_{23}^2$$

From $j=4$

$$U_{14} = \frac{A_{14}}{D_{11}}$$

$$U_{24} = \frac{A_{24} - U_{12}D_{11}U_{14}}{D_{22}}$$

$$U_{34} = \frac{A_{34} - U_{13}D_{11}U_{14} - U_{23}D_{22}U_{24}}{D_{33}}$$

$$D_{44} = A_{44} - D_{11}U_{14}^2 - D_{22}U_{24}^2 - D_{33}U_{34}^2$$

And so on. For purposes of efficient programming, it is worthwhile to introduce the auxiliary variable

$$L_{ji} \stackrel{\text{def}}{=} D_{ii}U_{ij}$$

Then, equivalent to the above, we have

From $j=1$

$$D_{11} = A_{11}$$

From $j=2$

$$L_{21} = A_{12}$$

$$U_{12} = L_{21}/D_{11}$$

$$D_{22} = A_{22} - L_{21}U_{12}$$

$L_{21} = D_{11}U_{12} \Rightarrow U_{12} = L_{21}/D_{11}$

$D_{22} = A_{22} - D_{11}U_{12}^2 = A_{22} - D_{11}U_{12}U_{12}$

$A_{22} - D_{11}\,L_{21}/D_{11}\,U_{12} = A_{22} - L_{21}U_{12}$

$$L_{31} = A_{13}$$

$$L_{32} = A_{23} - U_{12}L_{31}$$

$$U_{13} = L_{31}/D_{11}$$

$$U_{23} = L_{32}/D_{22}$$

$$D_{33} = A_{33} - L_{31}U_{13} - L_{32}U_{23}$$

$$L_{41} = A_{14}$$

$$L_{42} = A_{24} - U_{12}L_{41}$$

c

$$L_{43} = A_{34} - U_{13}L_{41} - U_{23}L_{42}$$

$$U_{14} = L_{41}/D_{11}$$

$$U_{24} = L_{42}/D_{22}$$

$$U_{34} = L_{43}/D_{33}$$

$$D_{44} = A_{44} - L_{41}U_{14} - L_{42}U_{24} - L_{43}U_{34}$$

And so on. Summarizing, for $j = 1, 2, \ldots, n$

$$L_{ji} = A_{ij} - \sum_{k=1}^{i-1} U_{ki}L_{jk}, \qquad 1 \leq i \leq j - 1$$

$$U_{ij} = L_{ji}/D_{ii}$$

$$D_{jj} = A_{jj} - \sum_{i=1}^{j-1} L_{ji}U_{ij}$$

Observe that no additional storage besides that needed for $\boldsymbol{A}$ is necessary in the torization process if $\boldsymbol{A}$ is overwritten by $\boldsymbol{U}$ and $\boldsymbol{D}$, viz.,

For $i = 2, 3, \ldots, j - 1$

$$A_{ij} \leftarrow A_{ij} - \sum_{k=1}^{i-1} A_{ki}A_{kj}$$

For $i = 1, 2, \ldots, j - 1$

$$T \leftarrow A_{ij}$$

$$A_{ij} \leftarrow T/A_{ii} = U_{ij}$$

$$A_{jj} \leftarrow A_{jj} - TA_{ij} = D_{jj}$$

This is the procedure coded in subroutine FACTOR in DLEARN. Additionally, FACTOR takes account of the profile storage of the coefficient matrix. The indexing necessary to account for the profile somewhat complicates the procedure but is neces- ry to achieve optimal efficiency. Figures 11.2.1(a) and 11.2.1(b) are presented as an aid for understanding the indexing and one-dimensional storage of the coefficient matrix in FACTOR. Note that if $A_{ii} = 0$, subroutine FACTOR skips the operations in the second do loop.

Accounting for the sparseness of A

Consider an original 8×8 A

Skyline

$$
\begin{bmatrix}
X & X & O & O & O & O & O & X \\
 & X & X & O & X & O & O & O \\
 & & X & X & X & O & O & O \\
 & & & X & O & X & O & X \\
 & & & & X & X & O & O \\
 & \text{Sym} & & & & X & X & O \\
 & & & & & & X & X \\
 & & & & & & & X
\end{bmatrix}
$$

Careful examination of the process just described will confirm that the process of factoring A the terms above the "skyline" are never touched they need not be stored and do not do operations up there –

Therefore

In real problems the area above the skyline is very large so you want to account explicitly for this.

Standard way to do this is to store A in a big vector and use a "vector of pointers" to know where things are.

ALHS ← The big vector , IDIAG ← Pointers to diagonal terms for that column (last term in ALHS for it)

Stored column by Coclumn (6)

For the example matrix

ALHS(1) = A_{11}	ALHS(13) = A_{56}	IDIAG(1) = 1
(2) = A_{12}	(14) = A_{66}	(2) = 3
(3) = A_{22}	(15) = A_{67}	(3) = 5
(4) = A_{23}	(16) = A_{77}	(4) = 7
(5) = A_{33}	(17) = A_{18}	(5) = 11
(6) = A_{34}	(18) = A_{28}	(6) = 14
(7) = A_{44}	(19) = A_{38}	(7) = 16
(8) = A_{25}	(20) = A_{48}	(8) = 24
(9) = A_{35}	(21) = A_{58}	
(10) = A_{45}	(22) = A_{68}	
(11) = A_{55}	(23) = A_{78}	
(12) = A_{46}	(24) = A_{88}	

Of course when you then write the procedures you have to understanding how to use the pointers and vector to keep track of the operations ← See Chapt. 11 of Hughes

We will do a particular one in our pseudo code. It will look convenient (although not necessarily the absolute most efficient – To do that requires detailed consideration of how machine handles data, etc.)

Direct equation solving by the frontal technique -
Ref - E. Hinton, D.R.J. Owen, Finite Element Programming, Chapt. 8.

- As other direct methods it is simply Gauss elimination -
- Specifically devised for F.E. calc. -

Idea is simple - assemble elements and eliminate variables as you go - As soon as all coefficients of an eq. are completely assembled - eliminate the variable -

- All that is "active" at a particular time is the upper triangle of all eq. that have had contributions put in, but not yet complete and reduced - these d.o.f. are called the front - ← # is front width - it changes as you go - max. size needed governed by maximum front width.

those dof in are "active"
those dof not yet in "inactive
those dof that have been eliminated - "deactivated"

Preliminaries to the process -

- number equations
- Order elements
- scheme to know when all entries for a dof are in and if it can be eliminated
- booking keeping system to keep track of whats in, whats done, whats to come

Bookkeeping -
- position in front where each dof of an element gets assembled - destination vector
- variable location array - where equations are and when they aren't - including when they are fully assembled
- active variables - variables currently in front - "front"

Copy pages 176, 177

Frontal Solution

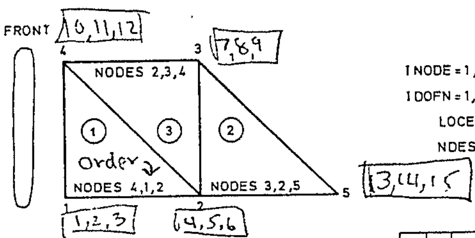
NDEST (KDOFE) → the assembly position for the golbal dof. # associated with that elemental dof.

LOCEL (KDOFE) → global dof. # associated with elemental dof. - made neg. when it can be eliminated.

NACVA (IFROW) = List of dof currently in front, NFROW is the front size., note zero locations before NFRONT means something has been eliminated - always go to these next.

3 dof/node

FIG. 8.1



up to
15 allowed →
in front 2

element 1
in

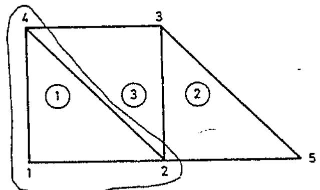
INITIALLY

INODE = 1, NNODE	0			0			0		
IDOFN = 1, NDOFN	0	0	0	0	0	0	0	0	0
LOCEL =	0	0	0	0	0	0	0	0	0
NDEST =	0	0	0	0	0	0	0	0	0

IFRON = 1, MFRON	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NACVA =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

NFRON = 0

FIG. 8.2



first
element
in

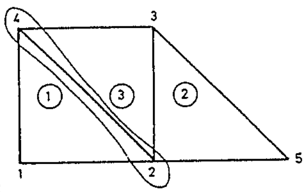
ELEMENT 1 - ASSEMBLY

INODE = 1, NNODE	1			2			3		
IDOFN = 1, NDOFN	1	2	3	1	2	3	1	2	3
LOCEL =	10	11	12	-1	-2	-3	4	5	6
NDEST =	1	2	3	4	5	6	7	8	9

IFRON = 1, MFRON	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NACVA =	10	11	12	1	2	3	4	5	6	0	0	0	0	0	0

NFRON

FIG 8.3



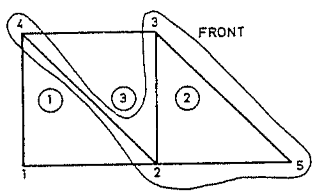
everything
for node
1 is in-
eliminate it

ELEMENT 1 - ELIMINATION

IFRON = 1, MFRON	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NACVA =	10	11	12	0	0	0	4	5	6	0	0	0	0	0	0

NFRON

FIG. 8.4



element 2 in

ELEMENT 2 - ASSEMBLY

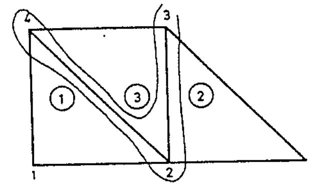
INODE = 1, NNODE	1			2			3		
IDOFN = 1, NDOFN	1	2	3	1	2	3	1	2	3
LOCEL =	7	8	9	4	5	6	-13	-14	-15
NDEST =	4	5	6	7	8	9	10	11	12

empty space filled

IFRON = 1, MFRON	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NACVA	10	11	12	7	8	9	4	5	6	13	14	15	0	0	0

NFRON

FIG 8.5



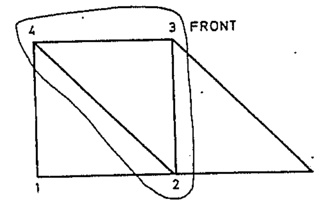
Can
elimate
dof at node 5

ELEMENT 2 - ELIMINATION

IFRON = 1, MFRON	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NACVA =	10	11	12	7	8	9	4	5	6	0	0	0	0	0	0

NFRON

FIG. 8.6



third
element in -
now have everything

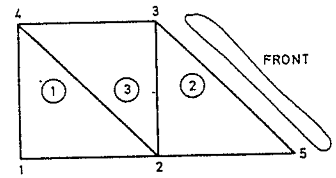
ELEMENT 3 - ASSEMBLY

INODE = 1, NNODE	1			2			3		
IDOFN = 1, NDOFN	1	2	3	1	2	3	1	2	3
LOCEL =	-4	-5	-6	-7	-8	-9	-10	-11	-12
NDEST =	7	8	9	4	5	6	10	11	12

IFRON = 1, MFRON	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NACVA =	10	11	12	7	8	9	4	5	6	0	0	0	0	0	0

NFRON

FIG. 8.7



ELEMENT 3 - ELIMINATION

IFRON = 1, MFRON	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NACVA =	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

NFRON = 0

FIGS. 8.1–8.7. "Housekeeping" operations for the assembly/elimination process.

7. Assembly

The purpose of the assembler class is to take the contributions of each force and stiffness contributor and assemble it into the global system. Assembler has two member functions called accept(...), the first takes a matrix and a list of degrees of freedom, the second takes a vector and a list of degrees of freedom. The first variation of this function corresponds to assembling a stiffness contributor and the second corresponds to assembling a force contributor. The list of degrees of freedom is used by the assembler to figure out if, and where to assemble each term in the matrix into the global matrix.

```
LinearSystemAssember::accept(Matrix k, List dofs)
{ // given a stiffness contributor stiffness matrix and list of associated DOF objects - add into global matrix
  int size = dofs->size(); // get size of matrix, same as size of list dofs
  for(int i = 0; i < size; i++){ // loop over the rows of the stiffness contributor
      DOF idof = dofs[i]; // get ith degree of freedom from the DOF object
      int ki = idof->EqNumber; // get global equation number for the current row from DOF object
      for(int j = 0; j < size; j ++){ // for the current row, loop over the columns of the local stiffness matrix
          DOF jdof = dofs[j]; // get jth degree of freedom from the DOF object
          int kj = jdof->EqNumber; // get global equation number for column from the DOF object
          if ( ki > kj ) // check that this term is in the upper diagonal of K, if not, skip it
              continue;
          if (jdof->Status = Free && idof->Status = Free){ // we have a potential upper triangle term
                                                          // see if it is by checking the status - both must be free
              K(ki,kj) += k(i,j); // add the local stiffness term to the correct location in the global matrix
          } else { // if both not free, then one or both are constrained, if one of them is constrained as
                  // Fixed (nonzero essential b'dry. cond.) and the other is free, the proper term must go
                  // into the load vector
              DOF cdof; // will need the value of the non-zero essential b'dry. cond. from the DOF object
              if( jdof->Status = Free and idof->Status = Fixed ) { // adds to the kj force term
                      cki = kj;
                      cdof = idof;
              } else if( (idof->Status = Free and jdof->Status = Fixed)){ // adds to the kj force term
                      cki = ki;
                      cdof = jdof;
              }
              // cki is the equation number, cdof is the constrained dof
              if(cdof)
                  f(cki) -= cdof->Value*k(i,j);
          }
      }
  }
}
```

8. Sparse Matrix

The SparseMatrix class implements a symmetric skyline storage scheme used for the global stiffness matrix.

Note: there are several other ways of implementing equivalent functionality. What is described in the following uses some specific C++ (and C) syntax and semantics which leads to a convenient implementation in which the functions factor() and backsub(...) can be written using standard array notation to access the elements of the stiffness matrix (K[i][j]) even though a skyline storage scheme is used.

SparseMatrix
setSize(int s) // sets the number of equations in the problem and initializes vector of pointers addNonZeroTerm(int i, int j) // accounts for the coupling of a nonzero term allocateMemory() // allocates memory to the matrix operator ()(int i, int j) : double // accesses the i,j entity of the stiffness matrix, i,j term must // be within the skyline factor() // factors the global stiffness matrix backsub(Vector f, Vector d) // does a back substitution of f on factored matrix, d returned
int Size; int *FirstEntry; double **K; // K is a pointer to a pointer of a double which is a stiffness matrix term

In this implementation the lower triangular portion of the matrix will be stored for convenience in accessing the elements. This does not alter the fact that we were adding only upper triangular terms. All that is required is to use the version of the SparseMatrix operator () that reverses the order. The variable FirstEntry is a vector that will be used to store the first non-zero column in each row.

In the skyline storage scheme, only the terms "under the shyline" need to be stored. We will use the following stiffness matrix to demonstrate what is done.

$$
\begin{bmatrix}
K_{00} & & & & & & & & & \\
K_{10} & K_{11} & & & \text{sym} & & & & & \\
0 & K_{21} & K_{22} & & & & & & & \\
0 & K_{31} & K_{32} & K_{33} & & & & & & \\
0 & 0 & K_{42} & K_{43} & K_{44} & & & & & \\
0 & 0 & K_{52} & K_{53} & K_{54} & K_{55} & & & & \\
K_{60} & K_{61} & K_{62} & K_{63} & K_{64} & K_{65} & K_{66} & & & \\
0 & 0 & 0 & 0 & 0 & K_{75} & K_{76} & K_{77} & & \\
0 & 0 & 0 & 0 & K_{84} & K_{85} & K_{86} & K_{87} & K_{88} & \\
0 & 0 & 0 & 0 & 0 & 0 & K_{96} & K_{97} & K_{98} & K_{99}
\end{bmatrix}
$$

The FirstEntry vector is to indicate the first column that a non-zero term appears in for each row. Note the first column is column 0:

$$
\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 0 & 5 & 4 & 6 \end{bmatrix}
$$

Before anything else the matrix must know the number of rows in the stiffness matrix. The size is simply ndof, the determination of which was stated before. The setSize member function is used to initialize the vector at the right size.

```
SparseMatrix::setSize(int s)
{
  Size = s // the size is equal to the variable ndof
  FirstEntry = new int[size]; // define the vector of that length
  for(int i = 0; i < s; i++) // this loop simply initializes the first column for each row to the main diagonal term
                   // since the main diagonal terms always exist, and you want to be sure thats reflected
      FirstEntity[i] = i;
}
```

In this function the ith entry in FirstEntity is set equal to i (each row must have an entry on the diagonal).

Next the structure of the matrix must be determined. That is the lowest non-zero column for each row. The procedure createGlobalSystem() was determining the basic coupling information while traversing the mesh. The addNonZeroTerm(int i, int j) member function actually carries out the updating of the FirstEntity vector based on the existence of the non-zero stiffness terms seen in createGlobalSystem(). The implementation of this for the skyline matrix is:

```
SparseMatrix::addNonZeroTerm(int i, int j)
{
 if(FirstEntry[j] > i) // will update only if it is a lower column number than the current lowest
      FirstEntry[j] = i;
}
```

This simply checks if the entry is outside the range currently stored in FirstEntry and if it is updates the appropriate item in FirstEntry.

Once addNonZeroTerm(...) is called for each non zero term in the stiffness matrix, the allocateMemory() member function must be called to allocate memory for the matrix and finish it's initialization. The pointer vector is also updated to point into the memory location of the stiffness matrix of the first term of a row assuming the row was filled. This allows the convenient indexing by i,j.

```
SparseMatrix::allocateMemory()
{ // first must figure out how much total memory is needed, sum of all matrix elements within skyline
 int i;
 int totalEntries = 0;
 for(i=0; i < size; i++) { // for each row add from the first non-zero column through the main diagonal
      totalEntries += i-FirstEntry[i]+1;
 }
 // allocate memory for K, the memory will store the stiffness matrix and the pointers in K will allow us
 // to address that memory by directly indicating the ith row and jth column
 K = new double *[Size];
 double *mem = new double[totalEntries] // allocate vector of memory to store the entire stiffness matrix
 // set up the pointers so that can access as K[i][j]
 int currentDiag = -1;
 for (int i = 0; i < size; i++){
      currentDiag += i - FirstEntry[i]+1;
      K[i] = &(mem[currentDiag-i]) // setting the pointer for ith row to the location where the a non-zero
                                   // term would be stored if the 0th column was non-zero
                                   // by doing this the desired Kij term is easily found by going j positions
                                   // from this location in memory
 }
}
```

The storage for our matrix is actually one large vector in which the elements are stored in the order (assuming the example matrix shown earlier):

$$[K_{00}, K_{10}, K_{11}, K_{21}, K_{22}, K_{13}, K_{32}, K_{33}, K_{42}, K_{43}, K_{44}, etc]$$

The final part of the allocateMemory() member function is setting up an array of pointers so that we can conveniently access the entries in K using the K[i][j] notation (this portion of the code is assuming we are coding in C++ or C (with minor changes)). Without going into the full details of how and why this works, what needs to be done is to allocate an array of pointers into K and set each pointer K[i] to point to where in memory the element 0 of the corresponding row is (or would be if it existed). This way to get the desired i,j stiffness matrix term by simply going j locations past where K[i] pointed into the memory.

$$K[0]\; K[1]\; K[2] \qquad K[3] \qquad K[4]$$

$$[K_{00}, K_{10}, K_{11}, K_{21}, K_{22}, K_{31}, K_{32}, K_{33}, K_{42}, K_{43}, K_{44}, etc]$$

Using this type of setup the element access operator (int i, int j) can be written as:

```
double SparseMatrix::operator()(int i, int j)
{ return K[i][j]; }
```

If we desire it to appear (from the point of view of anyone using the SparseMatrix class) that we are storing the upper diagonal of the matrix then this should be written as:

```
double SparseMatrix::operator()(int i, int j)
{ return K[j][i]; }
```

This is the one we would need to use with our current assembly operator.

9. Equation Solving

The two routines that follow implement the Crout solver discussed in class using our current storage structures.

```
void SparseMatrix::factor()
{
  int i,j,r;
  int mj,mi,mm;
 double *G = new double[Size]; // temporary vector of size = Size
  for (j=1; j != Size; j++){
       if (K[j][j] <= 0.0) // system is suppose to be positive definite, so this should not happen
                            // give an error message if it does
            cerr << "factor: Initial Negative or Zero Diagonal Term\n";
       mj = FirstEntry[j]; // get column of first entry in j'th row in K
       G[mj] = K[j][mj]; // initialize G
       for (i= mj+1; i <= j-1; i++){ // loop over the remainder of entries in j'th row
            G[i] = K[j][i];
            mi = FirstEntry[i]; // get column of first entry in i'th row of K
            mm = (mi >= mj) ? mi : mj; // get maximum of mi and mj
            for (r = mm; r <= i-1; r++) // loop over columns from mm to diagonal
                 G[i] -= K[i][r]*G[r];
       }
       for (i=mj; i <= j-1; i++) // loop mj to diagonal
            K[j][i] = G[i]/K[i][i];
       for (r = mj; r <= j-1; r++) // loop from mj to diagonal
            K[j][j] -= K[j][r]*G[r];
  }
  delete G;
}
```

```
void SparseMatrix::backsub(double *f, double *d)
// f and d are vectors of size = Size, f has the RHS, d will hold the solution
{
  int i,j;
  int mi;
  for (i = 0; i != Size; i++){ // initialize d to equal f
        d[i] = f[i];
  }
  for (i = 1; i != Size; i++){ // loop over rows of K
        mi = FirstEntry[i]; // get column of first entry in row i
        for (j= mi ; j <= i-1; j++) // loop over entries in row i
              d[i] -= K[i][j]*d[j];
  }
  for (i=0; i!= Size; i++) // loop over diagonal terms
        d[i] /= K[i][i];
  for (i= Size - 1; i >= 1 ; i--){ // loop backwards over rows
        mi = FirstEntry[i]; // get column of first entry in i'th row
        for (j = mi; j <= i-1; j++){ // loop over entries in i'th row
              d[j] -= K[i][j]*d[i];
        }
  }
}
```

10. Recovering Secondary Variables

In order to recover secondary variables (such as stress and strain) we must have already solved the global system and placed the results back into the correct DOF objects on the mesh (set the Value field of each DOF that had a Free status). In this manner the DOF object values represent a complete vector of the primary variables, both solved for and given (essential boundary conditions).

Once this is done, the calculation of secondary variables is fairly straightforward. This procedure will be explained without introducing any new functionality to the existing classes to keep things simple. In reality we would probably want to add some functionality to and perhaps add another class to do this.

In order to calculate strain from displacement, we need to calculate $u_{i,j}$. This can be calculated using the ShapeFunction class member functions dNds and sfdofs. Remember that dNds(pt) gives:

$$\begin{bmatrix} N_{1,r}(pt) & N_{2,r}(pt) & N_{3,r}(pt) & \dots & N_{n,r}(pt) \\ N_{1,s}(pt) & N_{2,s}(pt) & N_{2,s}(pt) & \dots & N_{n,s}(pt) \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} \frac{\partial}{\partial r} N_i \\ \frac{\partial}{\partial s} N_i \end{bmatrix}$$

and sfdofs returns the DOF's for the shape function in the corresponding order where each row contains the DOF objects for the two displacement components at the corresponding node (see example earlier in the notes). Note that each of these DOFs now has a value. If we take the above matrix and create a new one with the values of the DOFs denoting entries in the first column u_i and the second column v_i (to stand for the u and v components of displacement) and multiply it by what is returned by ShapeFunction::dNds we get something very useful, the derivatives of the displacement with respect to the local coordinate system. With this, and information from the mapping class, the derivatives with respect to the global coordinate system can be found and the secondary variables calculated.

$$\begin{bmatrix} \frac{\partial u}{\partial r} & \frac{\partial v}{\partial r} \\ \frac{\partial u}{\partial s} & \frac{\partial v}{\partial s} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial r}N_i \\ \frac{\partial}{\partial s}N_i \end{bmatrix} \begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \\ \dots & \dots \\ u_n & v_n \end{bmatrix}$$