Peter Cottle ***SID 19264824***

ME 280A – 12/10/2011

Homework Assignment #7

Introduction:

This homework project will serve as the only foray into 3D finite element formulations of statically loaded structures. Because of the complexity of this homework project, only the 3D mesher will be programmed in MATLAB; the rest of the work for the finite element solution will be presented in a theoretical manner.

3D finite element formulations are the cornerstone of the finite element industry and serve as the validation method for the majority of 3D structures in production today. Because displacement is an unknown 3D vector, there are three unknowns per node. With 8 nodes per element in the most basic formulations, this leads to 24 unknowns per element. As one can imagine, the sheer size of the final stiffness matrix becomes almost unbearable without domain decomposition or iterative solvers. This makes implementation difficult for even a graduate level class; consequently, only the outline of implementation for 3D will be presented here.

Objectives:

In this homework we have only one programming challenge: to implement a three dimensional mesher for the given pipe structure and construct the connectivity map matrix. Our mesher will be constrained to making these pipe shapes with various radii and lengths, primarily because full three dimensional meshers that can take on _any_ shape are licensed at several dozens of thousands of dollars per year. However, even this simple pipe shape is not as easy as one might think. The Findings section will present the mesher that I have implemented.

The rest of the homework will lay out a complete framework for the finite element formulation in three dimensions. The instructions are to:

- Develop the weak form for the stress equation presented in the homework.
  - Define the approximation function spaces.
- Develop the weak form with the penalty method as well.
  - Define the approximation function spaces.
- Define the stiffness matrices and load vectors explicitly in terms of trilinear basis functions, material properties, and loading functions.
  - Describe how the global stiffness matrix and load vectors are assembled from the element matrices (step by step).
- Do the same as above but for the Penalty method formulation.
  - Be explicit about the loading on surface elements versus the loading on entire element bodies.

- Indicate how to solve the problem with the implemented mesh generator and its output. Estimate the Galerkin Subspatial error in the solution, especially how to determine the constants.
- Characterize the geometric error as a function of Nr, N_theta, Ns.
- If one used the conjugate gradient solver, how many operations would be needed for Nr, N_theta, Ns?
- Consider the time transient case
    - Develop the weak form for:
        - Implicit finite difference for the time
        - Explicit finite difference for the time
        - Explicit finite difference for the time + lumped mass approximation

As one can see, that is a lot of work. The "Analytical" section will outline all of the analytical work, and the "Implementation" section will show the mesher.

Analytical:

Weak Form:

First, we define the loading function as:

$$\rho \mathbf{b} = f$$

Which gives us our starting equation of:

$$\nabla \cdot (E \vdots \nabla u) + f = 0$$

We define the stress to further simplify:

$$E \vdots \nabla u = \underline{\sigma}$$

Obtaining:

$$\nabla \cdot \underline{\sigma} + f = 0$$

Test function:

$$(\nabla \cdot \underline{\sigma} + f) \cdot v = 0$$

Integrate:

$$\int_{\Omega} (\nabla \cdot \underline{\sigma} + f) \cdot v d\Omega = 0$$

Use the equivalent of the "trick" in 3D space:

$$\nabla \cdot (\underline{\sigma} \cdot v) = (\nabla \cdot \underline{\sigma})v + \nabla v \vdots \underline{\sigma}$$

To obtain:

$$\int_\Omega \nabla \cdot (\underline{\sigma} \cdot v) d\Omega - \int_\Omega \nabla v \vdots \underline{\sigma} d\Omega$$
$$+ \int_\Omega f \cdot v d\Omega = 0$$

Then use the divergence theorem on the first term to obtain:

$$\int_\Gamma \underline{\sigma} \cdot \hat{n} \cdot v d\Gamma - \int_\Omega \nabla v \vdots \underline{\sigma} d\Omega$$
$$+ \int_\Omega f \cdot v d\Omega = 0$$

Rearrange to obtain the weak form:

$$\int_\Gamma \underline{\sigma} \cdot \hat{n} \cdot v d\Gamma + \int_\Omega f \cdot v d\Omega = \int_\Omega \nabla v \vdots \underline{\sigma} d\Omega$$

And we can reverse substitute to obtain:

$$\int_\Gamma \underline{\sigma} \cdot \hat{n} \cdot v d\Gamma + \int_\Omega f \cdot v d\Omega + =$$
$$\int_\Omega \nabla v \vdots E \vdots \nabla u d\Omega$$

Also, noting the more general form of traction forces and body loading forces, we obtain the following by substituting for the stress dot normal term:

$$\int_\Gamma t \cdot v d\Gamma + \int_\Omega f \cdot v d\Omega + =$$
$$\int_\Omega \nabla v \vdots E \vdots \nabla u d\Omega$$

Which is effectively Equation 9.1 from the reader, also known as the general form.

Approximation Spaces :

We also must take care to define the approximation function spaces for this formulation. In particular, we must define the restrictions upon the set of functions within the weak form that we have derived. These three restrictions govern the selection:

$$\forall v =$$
$$\int_\Omega f \cdot v d\Omega < \infty$$
$$\int_{\partial\Omega} \sigma \cdot n \cdot v d\Omega < \infty$$
$$\int_\Omega \nabla v \vdots \sigma d\Omega < \infty$$

Where we use the equals sign for a relation, rather than the strict equality it implies. We now switch to Hilbertian Sobolev spaces to denote the restrictions upon the solution, traction, and test function. In summary, belonging to an H^1 function space simply denotes:

$$u \in H^1(\Omega) \;\; if \; ||u||^2_{H^1(\Omega)} \overset{\text{def}}{=}$$
$$\int_\Omega \frac{\partial u_i}{\partial x_j}\frac{\partial u_i}{\partial x_j} d\Omega + \int_\Omega u_i u_i d\Omega < \infty$$

Thus, our restriction on the force and traction functions becomes:

$$f \in L^2(\Omega)$$
$$t \in L^2(\Gamma_t)$$

And our stress and function restrictions become:

$$u \in H^1(\Omega)$$
$$\sigma \in L^2(\Omega)$$

And the restriction on the test function and solution with boundary conditions:

$$\forall v \in H^1(\Omega), v|_{\Gamma_u} = 0$$
$$u \in H^1(\Omega), u|_{\Gamma_u} = d$$

Essentially, we want to find a test function inside H^1 that is 0 at the boundary where the displacement is specified. Likewise, we want to find a solution also in H^1 that is the specified value at the boundary.

Weak Form with Penalty Method:

The addition of the Penalty term:

$$P^{\star} \int_{\Gamma_u} (u_0 - u) \cdot v d\Gamma$$

Into the weak form gets split up on both sides and becomes:

$$\int_{\Gamma} \underline{\sigma} \cdot \hat{n} \cdot v d\Gamma + \int_{\Omega} f \cdot v d\Omega + \int_{\Gamma} P^{\star} u_0 \cdot v d\Gamma =$$
$$\int_{\Omega} \nabla v \colon \underline{\sigma} d\Omega + \int_{\Gamma} P^{\star} u \cdot v d\Gamma$$

As expected from the earlier penalty term formulation in Homework #6. We can reverse substitute the stress and normal term to obtain this from the reader:

$$\int_{\Gamma} t \cdot v d\Gamma + \int_{\Omega} f \cdot v d\Omega + \int_{\Gamma} P^{\star} u_0 \cdot v d\Gamma =$$
$$\int_{\Omega} \nabla v \colon E \colon \nabla u d\Omega + \int_{\Gamma} P^{\star} u \cdot v d\Gamma$$

Approximation Spaces with Penalty Method:

Now with the penalty method, we must again choose the space of approximation functions for the Pstar value over the boundary. Our formulation also changes because the restrictions on v and u change, primarily because v now exists everywhere and u has no restrictions:

$$\forall v \in H^1(\Omega)$$
$$u \in H^1(\Omega)$$

This change is an immediate advantage of the penalty method. The solution u exists everywhere without restriction (but is forced into the displacement condition by the added stiffness at those nodes), and the test function also exists everywhere and will not reduce the size of the overall stiffness matrix. The other restrictions from the earlier approximation spaces section still hold however.

Finally, just to clarify on Pstar, it exists at the displacement boundary and is zero elsewhere:

$$P^{\star}|_{\Gamma_d} = P^{\star}$$
$$P^{\star}|_{\bar{\Gamma}_d} = 0$$

Stiffness Matrices, Load Vectors, etc:

The tricky thing with the 3D formulation of the finite element method is that each traditional matrix gets expanded because of the addition of two dimensions. These expansions can be difficult to keep track of, and consequently the explicit formulations for the stiffness matrices, load vectors, material properties, and loading functions are explained below.

Additionally, all of the basis functions and evaluations must be mapped into the zeta domain for Gaussian integration. This means that the chain rule must be applied to map all of the derivative terms of the shape functions, and deformation gradients (as well as Jacobians) will now appear throughout the integral. This is shown in the second half.

Starting again with the previous formulation:

$$\int_{\Gamma} t \cdot v d\Gamma + \int_{\Omega} f \cdot v d\Omega + \int_{\Gamma} P^{\star} u_0 \cdot v d\Gamma =$$
$$\int_{\Omega} \nabla v : E : \nabla u d\Omega + \int_{\Gamma} P^{\star} u \cdot v d\Gamma$$

We can substitute in the expressions for v and u to obtain the following in matrix form

$$\int_\Omega ([D]\{v\})^T[E]([D]\{u\})d\Omega = \int_\Omega \{v\}^T\{f\}d\Omega$$

$$\int_{\Gamma_t}\{v\}^T\{t\}dA + P^\star \int_{\Gamma_u}\{v\}^t\{d-u\}dA$$

Where the 6x3 D deformation tensor is defined as:

$$[D] = \begin{bmatrix} \frac{\partial}{\partial x_1} & 0 & 0 \\ 0 & \frac{\partial}{\partial x_2} & 0 \\ 0 & 0 & \frac{\partial}{\partial x_3} \\ \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} & 0 \\ 0 & \frac{\partial}{\partial x_3} & \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} & 0 & \frac{\partial}{\partial x_1} \end{bmatrix}$$

And the vectors:

$$\{u\} = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix}$$

$$\{f\} = \begin{Bmatrix} f_1 \\ f_2 \\ f_3 \end{Bmatrix}$$

$$\{t\} = \begin{Bmatrix} t_1 \\ t_2 \\ t_3 \end{Bmatrix}$$

The 3x24 phi matrix is defined as the following:

$$[\phi] \overset{\text{def}}{=}$$
$$\begin{bmatrix} \phi_1\phi_2\phi_3\phi_4\ldots\phi_N & 00000\ldots & 00000\ldots \\ 00000\ldots & \phi_1\phi_2\phi_3\phi_4\ldots\phi_N & 00000\ldots \\ 00000\ldots & 00000\ldots & \phi_1\phi_2\phi_3\phi_4\ldots\phi_N \end{bmatrix}$$

Further recognizing that the phi functions have associated A and B coefficients for the v and u functions gives us:

$$\int_\Omega ([D][\phi]\{b\})^T[E]([D][\phi]\{a\})d\Omega = \int_\Omega ([\phi]\{b\})^T\{f\}d\Omega +$$
$$\int_{\Gamma_t} ([\phi]\{b\})^T\{t\}dA + P^\star \int_{\Gamma_u} ([\phi]\{b\})^T\{d - ([\phi]\{a\})\}dA$$

Next, these functions must be transformed into the zeta world. In order to do this, we first introduce the deformation gradient:

$$F \overset{\text{def}}{=}$$
$$\begin{bmatrix} \frac{\partial x_1}{\partial \zeta_1} & \frac{\partial x_1}{\partial \zeta_2} & \frac{\partial x_1}{\partial \zeta_3} \\ \frac{\partial x_2}{\partial \zeta_1} & \frac{\partial x_2}{\partial \zeta_2} & \frac{\partial x_2}{\partial \zeta_3} \\ \frac{\partial x_3}{\partial \zeta_1} & \frac{\partial x_3}{\partial \zeta_2} & \frac{\partial x_3}{\partial \zeta_3} \end{bmatrix}$$

Thus the following mapping can be defined:

$$\begin{Bmatrix} dx_1 \\ d_x2 \\ dx_3 \end{Bmatrix} = \begin{bmatrix} \frac{\partial x_1}{\partial \zeta_1} & \frac{\partial x_1}{\partial \zeta_2} & \frac{\partial x_1}{\partial \zeta_3} \\ \frac{\partial x_2}{\partial \zeta_1} & \frac{\partial x_2}{\partial \zeta_2} & \frac{\partial x_2}{\partial \zeta_3} \\ \frac{\partial x_3}{\partial \zeta_1} & \frac{\partial x_3}{\partial \zeta_2} & \frac{\partial x_3}{\partial \zeta_3} \end{bmatrix} \begin{Bmatrix} d\zeta_1 \\ d\zeta_2 \\ d\zeta_3 \end{Bmatrix}$$

With this mapping, the partial derivatives of the deformation tensor must be mapped. The first column becomes:

$$\begin{bmatrix} \frac{\partial}{\partial \zeta_1}\frac{\partial \zeta_1}{\partial x_1} + \frac{\partial}{\partial \zeta_2}\frac{\partial \zeta_2}{\partial x_1} + \frac{\partial}{\partial \zeta_3}\frac{\partial \zeta_3}{\partial x_1} \\ 0 \\ 0 \\ \frac{\partial}{\partial \zeta_1}\frac{\partial \zeta_1}{\partial x_2} + \frac{\partial}{\partial \zeta_2}\frac{\partial \zeta_2}{\partial x_2} + \frac{\partial}{\partial \zeta_3}\frac{\partial \zeta_3}{\partial x_2} \\ 0 \\ \frac{\partial}{\partial \zeta_1}\frac{\partial \zeta_1}{\partial x_3} + \frac{\partial}{\partial \zeta_2}\frac{\partial \zeta_2}{\partial x_3} + \frac{\partial}{\partial \zeta_3}\frac{\partial \zeta_3}{\partial x_3} \end{bmatrix}$$

The rest of the columns are mapped with the chain rule in a similar manner. The full expressions can be viewed on Page 81 of the reader.

The 3x24 phi matrix then becomes:

$$[\hat{\phi}] \stackrel{\text{def}}{=}$$
$$\begin{bmatrix} \hat{\phi}_1\hat{\phi}_2\hat{\phi}_3\hat{\phi}_4 \ldots \hat{\phi}_N & 00000\ldots & 00000\ldots \\ 00000\ldots & \hat{\phi}_1\hat{\phi}_2\hat{\phi}_3\hat{\phi}_4 \ldots \hat{\phi}_N & 00000\ldots \\ 00000\ldots & 00000\ldots & \hat{\phi}_1\hat{\phi}_2\hat{\phi}_3\hat{\phi}_4 \ldots \hat{\phi}_N \end{bmatrix}$$

The phi functions are defined by the Lagranian Polynomials. They can also be defined by taking all the plane intersection equations that do not contain the node in equation. For example, phi_1 is defined as:

$$\hat{\phi}_1 = \frac{1}{8}(1-\zeta_1)(1-\zeta_2)(1-\zeta_3)$$

The rest of the phi function definitions can be viewed on page 79 of the class reader.

Thus, our stiffness integral becomes:

$$\int_\Omega \frac{d\phi_i}{dx} E \frac{d\phi_j}{dx} dx = \int_{-1}^{1} \frac{d\hat{\phi}_i}{d\zeta} E \frac{d\hat{\phi}_j}{d\zeta} \frac{1}{J} d\zeta$$

Thus we can finally expression our stiffness matrices and load vectors in terms of a Gaussian summation over the zeta domain:

$$K_{ij}^e = \sum_{q=1}^{9}\sum_{r=1}^{9}\sum_{s=1}^{9} w_q w_r w_s ([\hat{D}\{\hat{\phi}_j\}])^T [\hat{E}]([\hat{D}\{\phi_j\}])|F|$$

And with **Penalty Term:**

$$K_{ij}^e = \sum_{q=1}^{9}\sum_{r=1}^{9}\sum_{s=1}^{9} w_q w_r w_s ([\hat{D}\{\hat{\phi}_j\}])^T [\hat{E}]([\hat{D}\{\phi_j\}])|F| +$$
$$\sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r P^{\star}[\hat{\phi}_i]^T\{\hat{\phi}_j\}|F_s|$$

Where the Fs is the surface Jacobian for the traction term.

We can also express the loading vector now with a Gaussian summation:

$$R_i^e = \sum_{q=1}^{9}\sum_{r=1}^{9}\sum_{s=1}^{9} w_q w_r w_s \{\hat{\phi}_i\}^T\{f\}|F| +$$
$$\sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r [\hat{\phi}_i]^T\{t\}|F_s|$$

And with **penalty term:**

$$R_i^e = \sum_{q=1}^{9}\sum_{r=1}^{9}\sum_{s=1}^{9} w_q w_r w_s \{\hat{\phi}_i\}^T\{f\}|F| +$$
$$\sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r [\hat{\phi}_i]^T\{t\}|F_s| + \sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r P^{\star}[\hat{\phi}_i]^T\{\hat{u}\}|F_s|$$

Finally, we have an expression for both the right hand and left hand side of the Ka=R equation.
Lastly, we must remember to map the material properties correctly:

$$E(x) = \hat{E}(x(\zeta_1, \zeta_2, \zeta_3))$$

And the traction loading term:

$$t(x) = \hat{t}(x(\zeta_1, \zeta_2, \zeta_3))$$

Assembly Steps:

After doing a triple summation 9-point Gaussian integration, we will obtain a 24x24 matrix for the stiffness matrix:

$$K_{ij}^{e} = \begin{bmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \dots & K_{1,24} \\ K_{2,1} & \ddots & & & \\ K_{3,1} & & \ddots & & \\ \vdots & & & \ddots & \\ K_{24,1} & & & & K_{24,24} \end{bmatrix}$$

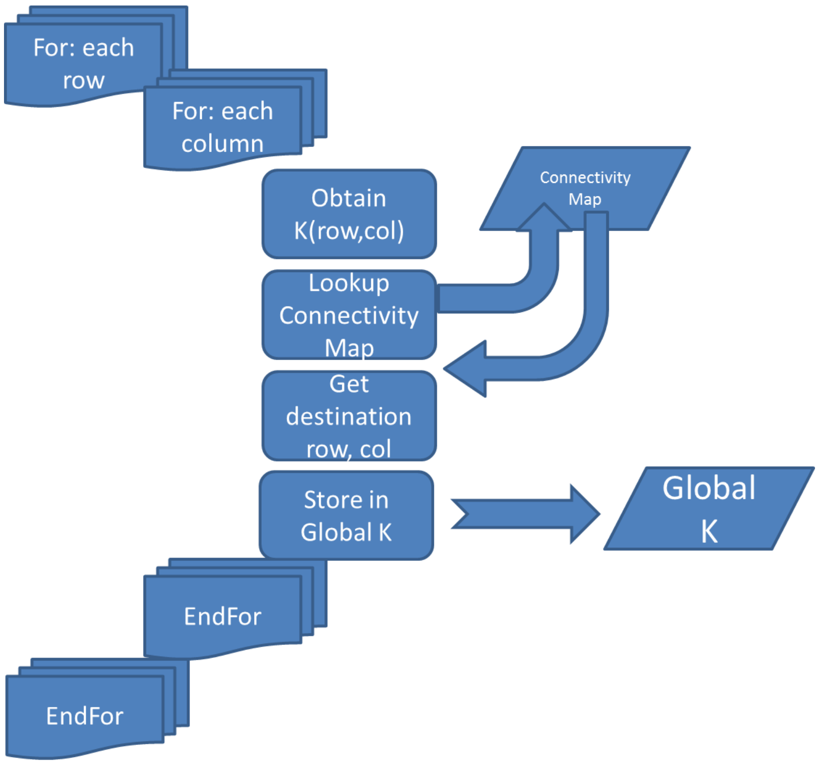
This 24x24 matrix corresponds to the 24 unknowns per element (3 per 8 nodes):

$$K_{ij}^{e}\{a\} = \begin{bmatrix} K_{1,1} & K_{1,2} & K_{1,3} & \dots & K_{1,24} \\ K_{2,1} & \ddots & & & \\ K_{3,1} & & \ddots & & \\ \vdots & & & \ddots & \\ K_{24,1} & & & & K_{24,24} \end{bmatrix} \begin{bmatrix} u_x^1 \\ u_y^1 \\ u_z^1 \\ u_x^2 \\ u_y^2 \\ u_z^2 \\ \vdots \\ u_z^{24} \end{bmatrix}$$

Similarly, the R loading vector corresponds to the loading at each node in each direction:

$$R_j^e = \begin{bmatrix} r_x^1 \\ r_y^1 \\ r_z^1 \\ r_x^2 \\ r_y^2 \\ r_z^2 \\ \vdots \\ r_z^{24} \end{bmatrix}$$

However, each of these per-element vectors must be assembled into their corresponding global vectors. For this, we must use the **connectivity map** in order to assemble each individual piece correctly. Flow Chart for stiffness assembly:



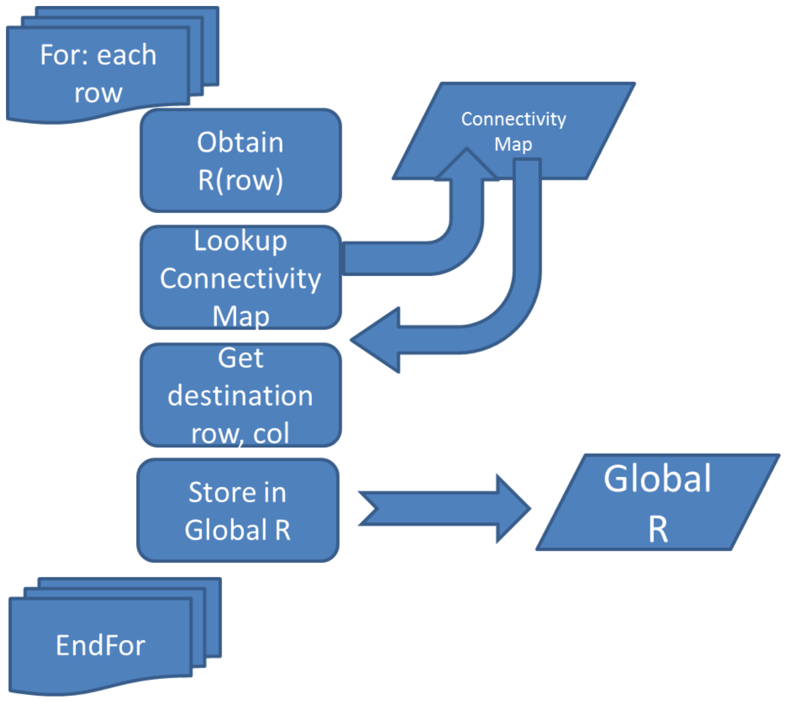
Code Layout:

- For each row
  - For each column
    - Obtain the local stiffness value at this row and column.

        - Use the connectivity map to determine which **global** node this **local** node corresponds to
        - Determine the global row and column based on this connectivity information
        - Add this local stiffness value to the global stiffness matrix
    - End for
- End for

Assembly for R vector:

The R vector is assembled in a similar way, except for all assembly only happens in one dimension:



Code Layout:

- For each row
        - Obtain the local loading value at this ro.
        - Use the connectivity map to determine which **global** node this **local** node corresponds to
        - Determine the global row based on this connectivity information
        - Add this local loading value to the global loading vector
- End for

The mesher section of this report will further detail the exact dimensions and information stored in the connectivity map matrix.

Penalty Method, Loading, and Surface Integrals:

It is important to note what each term in the right hand side represents. Going back to the Gaussian integration form in the zeta domain:

$$R_i^e = \sum_{q=1}^{9}\sum_{r=1}^{9}\sum_{s=1}^{9} w_q w_r w_s \{\hat{\phi}_i\}^T \{f\}|F| +$$

$$\sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r [\hat{\phi}_i]^T \{t\}|F_s| + \sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r P^{\star} [\hat{\phi}_i]^T \{\hat{u}\}|F_s|$$

The first term:

$$R_i^e = \sum_{q=1}^{9}\sum_{r=1}^{9}\sum_{s=1}^{9} w_q w_r w_s \{\hat{\phi}_i\}^T \{f\}|F|$$

represents the three dimensional integral of the loading force over the volume of the element. This integral will apply to **every element**; it may be zero for some elements if the loading force is zero at those locations, but this integral is fundamental to the finite element method.

To evaluate this expression, we vary zeta1, zeta2, and zeta3 over their entire ranges to capture all the behavior of the density and loading functions throughout the volume integral.

The second term:

$$\sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r [\hat{\phi}_i]^T \{t\}|F_s|$$

represents any **traction** loading terms on the **surface** of the integral. This integral only exists on **surfaces** of elements; consequently, it uses the Surface Deformation Gradient. This translates into the **surface jacobian** when the determinant is evaluated.

This expression is evaluated by fixing one of the zeta values. By fixing one of the zeta values to either +1 or -1, we will essentially evaluate this integral along one side of the master cube element. This side should be the size where this traction loading force exists. This expression **will be zero for all elements that do not have a surface** on the outside of the structure. For those that do have a surface on the outside of the structure, this expression will be evaluated for every side of the element that contains a traction loading term.

Furthermore, I would like to note that if we had any line traction loading, a very similar expression would be produced that only has one Gaussian summation. This would essentially be an integral along a line of the master cube element. Point loading is also possible as well, where the expression just simplifies to one evaluation at a node. These types of loading can be found in popular CAD and FEM software.

Lastly, the penalty term:

$$\sum_{q=1}^{9}\sum_{r=1}^{9} w_q w_r P^{\star}[\hat{\phi}_i]^T\{\hat{u}\}|F_s|$$

also exists only on surfaces where there is a specified displacement. It is zero elsewhere. This expression will be zero for all elements that do not share a surface where the displacement is specified. Please note that there could be specified displacements *within* a structure if needed, so the exterior / interior differentiation is no longer made.

Total Solution:

Now that we have expressions for every part of the 3D finite element formation, we are ready to solve. The overflow outline is below:

- For every element:
  - Compute the stiffness matrix
    - Use the triple summation expression given above
    - If these nodes have a specified displacement and we are using the penalty method, make sure to include that double summation term as well
  - Compute the loading vector
    - This involves two expressions:
      - The loading force triple summation over the volume
      - The traction force double summation over the surface
    - And a possible third term of the penalty
      - The penalty double summation over the surface with the specified displacement. Here, the actual displacement value is used to multiply against Pstar
  - Assemble the stiffness matrix into the global stiffness matrix
    - This can be done with the algorithm I outlined above
  - Assemble the loading vector in a similar manner
- Finally, if you are not doing the penalty term:
  - Eliminate the rows of the stiffness matrix that correspond to specified displacements
  - Take the corresponding columns and multiply them by the specified displacement vector and move to the right hand side
- Solve the Ka=R problem:
  - Either take the inverse of the global stiffness matrix (bad idea) or:
  - Use a preconditioned conjugate gradient solver

- Once the displacements for each node are known, post process for the stress:
    - Calculate the dimensionless strain with:

$$\begin{Bmatrix} \sigma_{11}^h \\ \sigma_{22}^h \\ \sigma_{33}^h \\ 2\sigma_{12}^h \\ 2\sigma_{23}^h \\ 2\sigma_{13}^h \end{Bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1} & 0 & 0 \\ 0 & \frac{\partial}{\partial x_2} & 0 \\ 0 & 0 & \frac{\partial}{\partial x_3} \\ \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_1} & 0 \\ 0 & \frac{\partial}{\partial x_2} & \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} & 0 & \frac{\partial}{\partial x_1} \end{bmatrix} \begin{Bmatrix} \sum_{i=1}^8 u_{1i}^h \phi_i \\ \sum_{i=1}^8 u_{2i}^h \phi_i \\ \sum_{i=1}^8 u_{3i}^h \phi_i \end{Bmatrix}$$

    - Multiply this by the material properties to get the stresses:

$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{13} \end{Bmatrix} = \begin{bmatrix} E_{1111} & E_{1122} & \cdots & E_{1123} & \cdots \\ E_{2211} & \vdots & & \vdots & \\ \vdots & & \ddots & & \\ E_{2311} & & & E_{1212} & \\ \vdots & & & & \end{bmatrix} \begin{Bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ 2\epsilon_{12} \\ 2\epsilon_{23} \\ 2\epsilon_{13} \end{Bmatrix}$$

After this, we have now solved for each stress and strain at each node. We can display these stresses and strains with fancy color plots if needed.

Galerkin Subspatial Error:

We are also interested in the error of our solution. First we define our error norm as:

$$||u-w||^2_{E(\Omega)} \overset{\text{def}}{=}$$
$$\int_\Omega (\frac{du}{dx} - \frac{dw}{dx}) E (\frac{du}{dx} - \frac{dw}{dx}) dx$$

Thus, we can define the error of our solution as:

$$||u-u^h||_{E(\Omega)} \leq$$
$$C(u,p)h^{min(r-1,p)=\gamma}$$

where p is the polynomial order of our basis functions (which will be just one [linear] for our formulation in this homework project). R is the regularity of the solution, and C is a constant that is dependent on the actual problem. We would like to determine the C constant however (the p is already known of course) and the gamma constant. Here, the h is obviously the element size like in 1D. In order to find C, make one coarse mesh and the another successively finer mesh and apply the following steps:

$$||u-u^{h_1}||^2_{E(\Omega)} = 2(J(u^{h_1}) - J(u^h)) \approx C^2 h_1^{2\gamma}$$
$$||u-u^{h_2}||^2_{E(\Omega)} = 2(J(u^{h_2}) - J(u^h)) \approx C^2 h_2^{2\gamma}$$

Then solve for C:

$$C = \sqrt{\frac{2(J(u^{h_1}) - J(u^{h_2}))}{h_1^{2\gamma} - h_1^{2\gamma}}}$$

For the gamma constant:

$$\frac{||u-u^{h_2}||_{E(\Omega)}}{||u-u^{h_1}||_{E(\Omega)}} = (\frac{h_2}{h_1})^\gamma$$

Then a simple logarithmic operation can be performed to obtain gamma.

Geometric Error:

The geometric error as a function of the element division parameters:

$$N_\theta, N_s, N_r$$

Essentially equates to finding the three constants in the reduction in error over three successively finer meshes:

$$ah_1^{\alpha} + bh_1^{\beta} + c = J(u^{h_1})$$
$$ah_2^{\alpha} + bh_2^{\beta} + c = J(u^{h_2})$$
$$ah_3^{\alpha} + bh_3^{\beta} + c = J(u^{h_3})$$

And then finding the zeta to monitor the error:

$$\xi = \frac{2(J(u^{h_3}) - c)}{\int_{\Omega} \nabla u^{h_3}:E:\nabla u^{h_3} d\Omega}$$

This can be done three separate times, refining the mesh with only one parameter each time. Then the three different constants that represent the error can be determined.

CG Operations:

As discussed in class, conjugate gradient operations perform on the order of:

$$O(\sqrt{K}M)$$

Where K is the condition number of the matrix and M here is the number of **degrees of freedom** in the problem. Assuming we can get the condition number of our matrix down to around 1, the conjugate gradient operates with respect to the number of degrees of freedom in the problem. This is essentially the number of unknowns in the problem, which is three per node. If there are 8 nodes per element…

$$M_{elements} = N_r N_{\theta} N_s$$
$$CG_{operations} = M_{elements} \cdot 8 \cdot 3$$
$$O(CG_{solver}) = O(24M)$$

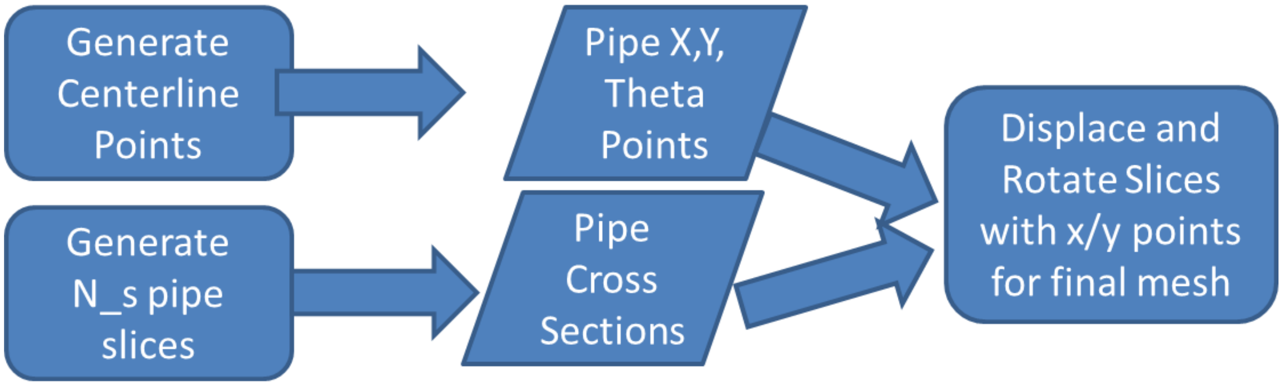
Hence, about 24 * M iterations to obtain a solution within tolerance. This is assuming **preconditioning** however. I omitted the P above for clarity.

Mesher Implementation:

Constructing the 3D mesh of the pipe structure was surprisingly easy once an elegant algorithm was constructed. The implementation of this algorithm will be outlined below:

Algorithm Overview:

The overall format of the algorithm is simple:


Generate Centerline Points
Pipe X,Y, Theta Points
Generate N_s pipe slices
Pipe Cross Sections
Displace and Rotate Slices with x/y points for final mesh

- First, the centerline points of the pipe (along with the angle of rotation) is generated.
    - These centerline points describe the rotation and position of each pipe cross section.
- Next, pipe cross sections are generated with the arch mesher from Homework #6 set to a full circle.
    - These pipe cross sections are all identical before rotation and displacement.
- Finally, these pipe cross sections are combined with the x and y centerline points to construct the full three dimensional mesh.
    - These is achieved by a two step process.
        - First, the pipe slice is rotated by an angle of 1-theta about the y axis. This effectively rotates the pipe cross section to the correction orientation. This is achieved with a rotation matrix.
        - The pipe cross section is then translated with homogenous coordinates and a translation matrix to the correct location.

Algorithm Details:

Cross Section Generator

The pseudo code and full implementation of the arch mesher (including the circle configuration) is specified in **Homework #6**. I will refer to this homework in order to determine how to generate the pipe cross sections. The main difference is that the end angle theta has to be set to 2*pi, which will both wrap the arch completely around in a circle and perform the connectivity at the seam. The rest is the same.

Centerline Point and Theta Generator

The centerline point generator essentially finds the x and y points along a circle of a certain radius of curvature. If these points are below zero, they are shifted over, because the pipe we are generating is essentially a circle with one half displaced by the diameter.

Pseudo Code:

```
Given: the radius of curvature and the s_step amount:

Generate an angle vector from -pi to pi, stepping by s_step

For I=1:length(angleVector):
   xPointHere = radius_curvature * cos(angleVector[i])

   if sin(thisAngle) > 0
      If the sin is above zero, we are now at the second half of the
      pipe
      Thus, displace x by 2 * radius_curvature
      X = x + 2*radius_curvature
   Endif
Endfor

Loop through each x point:
      If(x[i] <= radius_curvature):
            Calculate the y = sqrt(radius_curvature^2 - x^2)
            Theta = atan(y / x)
            Fix theta if positive or 0
      Else:
            Y = sqrt(radius^2-(radius*2-x)^2)
            Theta = atan(y / (radius*2-x))
      Endif
Endloop
```

Cross Section Rotations and Displacements

The cross section rotator essentially just rotates and displaces every pipe cross section to the desired location. It also generates the connectivity map by looping through each element. The matrices used are detailed blow:

Rotation matrix:

$$\begin{pmatrix} cos(\phi) & 0 & sin(\phi) \\ 0 & 1 & 0 \\ -sin(\phi) & 0 & cos(\phi) \end{pmatrix}$$

$$\phi = 2\pi - \theta$$

Translation matrix (Homogenous Coordinates):

$$\begin{pmatrix} cos(\phi) & 0 & sin(\phi) \\ 0 & 1 & 0 \\ -sin(\phi) & 0 & cos(\phi) \end{pmatrix}$$
$$\phi = 2\pi - \theta$$

Pseudo Code:

```
Given: the r_step, theta_step, s_step values

Slices = GENERATE_ALL_CROSS_SECTIONS()

Points = GET_CENTERLINE_POINTS()

Loop through EACH SLICE:
      For this slice, get the rotation amount:
      rotAmount = 2 * pi - thetaAtSlice

      Construct the rotation matrix in the y direction:

      Construct the translation matrix for the x and z amounts.

      For every point in cross section:
            Put into homogenous coordinates
            Rotate by matrix
            Translate by matrix
            Save result
      Endfor
Endloop

For every element in entire mesh:
      LLnode is sideCounter + (sliceCounter-1)*(numElementsInSlice+1) +
      (sCounter-1)*(numPointsInSlice)
      LR = LL + 1

      Rest of lower nodes are calculated in a similar matter

      For upper nodes: simply add the numPointsInSlice amount

      Draw this element’s connectivity with lines

      Save these node numbers into the connectivity map

endfor
```

Data Structure Results

The above code outputs two data structures: the connectivity map matrix and the global node coordinate matrix.

The **connectivity map** essentially relates the local node numbers to the global node numbers **for each element**. Hence, it has **M** element rows and has **8** columns for 8 of the each nodes in our trilinear approximation functions. A symbolic representation:

Connectivity Map					
Row	Col1	Col2	Col3	...	Col8
Element 1	Global node - LLL	Global node - LLR	Global node - LUR	...	Global Node – UUL
Element 2	Global node - LLL	Global node - LLR	Global node - LUR	...	Global Node – UUL
Element 3	Global node - LLL	Global node - LLR	Global node - LUR	...	Global Node – UUL
...					
Element M	Global node - LLL	Global node - LLR	Global node - LUR	...	Global Node - UUL

And the definition of the naming convention:

Letter	Zeta Value	Letter	Zeta Value
L__	z3 = -1	U__	z3=+1
_L_	z1 = -1	_U_	z2=+1
__L	z1 = -1	__R	z1=+1

A sample from the first mesh given below:

Connectivity Map								
Element	Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8
1	1	2	5	4	34	35	38	37
2	2	3	6	5	35	36	39	38
3	4	5	8	7	37	38	41	40
4	5	6	9	8	38	39	42	41
5	7	8	11	10	40	41	44	43

The **globalToCoordinate Matrix** simply maps each node number to a position vector with x, y, and zeta values. A symbolic representation:

Global Node to Coordinate Matrix			
	X	Y	Z
Node 1	X1	Y1	Z1
Node 2	X2	Y2	Z2
Node 3	X3	Y3	Z3
Node 4	X4	Y4	Z4
...	...	...	...

Node N	XN	YN	ZN

A sample from the first mesh given below:

Global Node to Coordinate Matrix			
	X	Y	Z
Node 1	-8	0	0
Node 2	-7.5	0	0
Node 3	-7	0	0
Node 4	-8.38197	1.175571	0
Node 5	-7.97746	1.469463	0
Node 6	-7.57295	1.763356	0

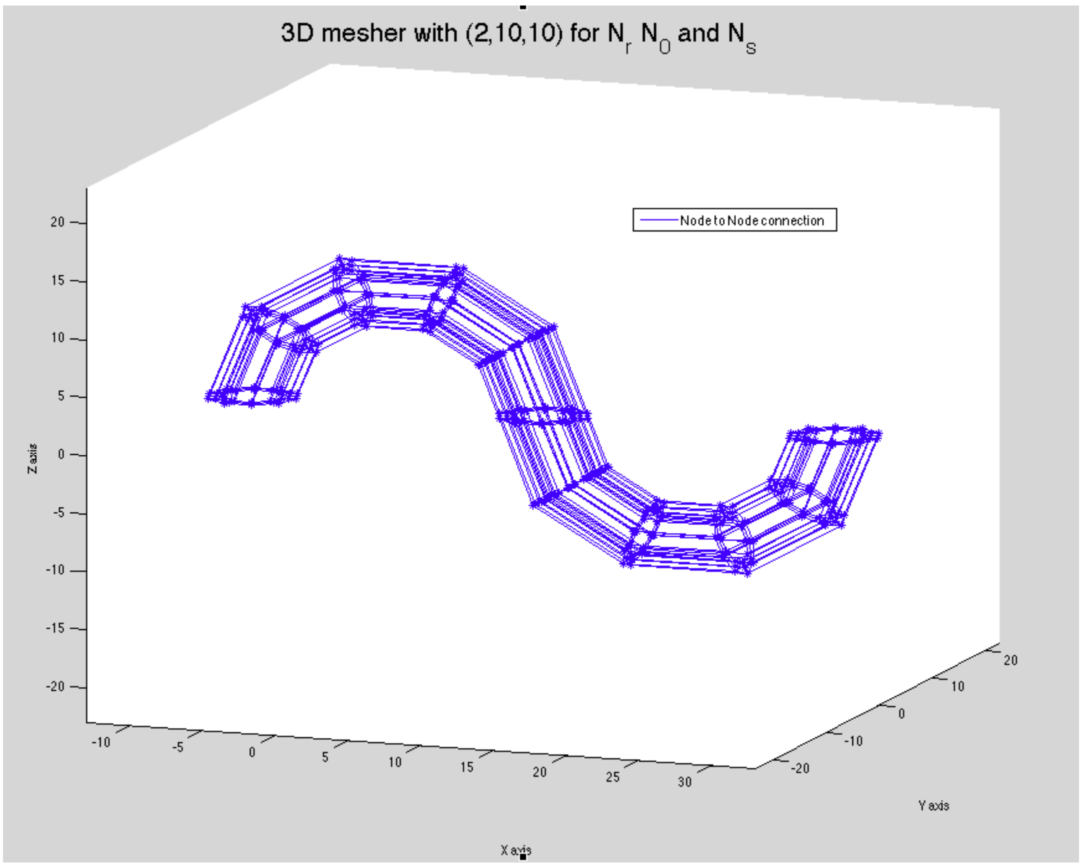
Mesher Findings

I am quite proud of the mesher that I have implemented. It is quite flexible and can be used to produce pipe meshes with various radii of curvature, inner and outer radii, end and start locations, and division parameters. Furthermore, because this algorithm uses my homework #6 algorithm for arch meshes, it can also generate "arch pipes" where there is a lengthwise section of the pipe missing.

Regardless, below are my plots for each mesh.

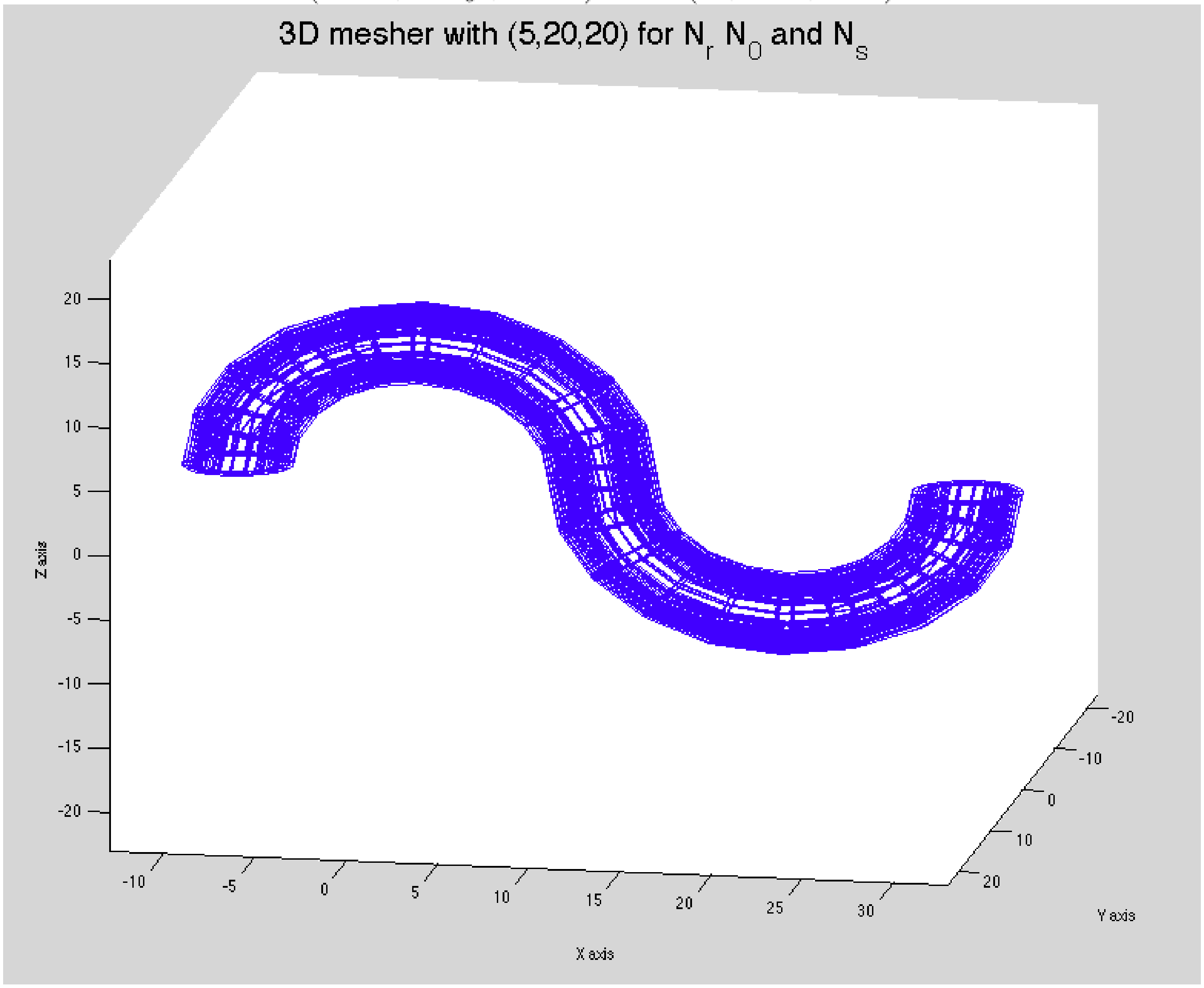
For this mesh, the node locations are shown with a star and the element connectivity is shown with lines:

$$(N_r, N_\theta, N_s) = (2, 10, 10)$$



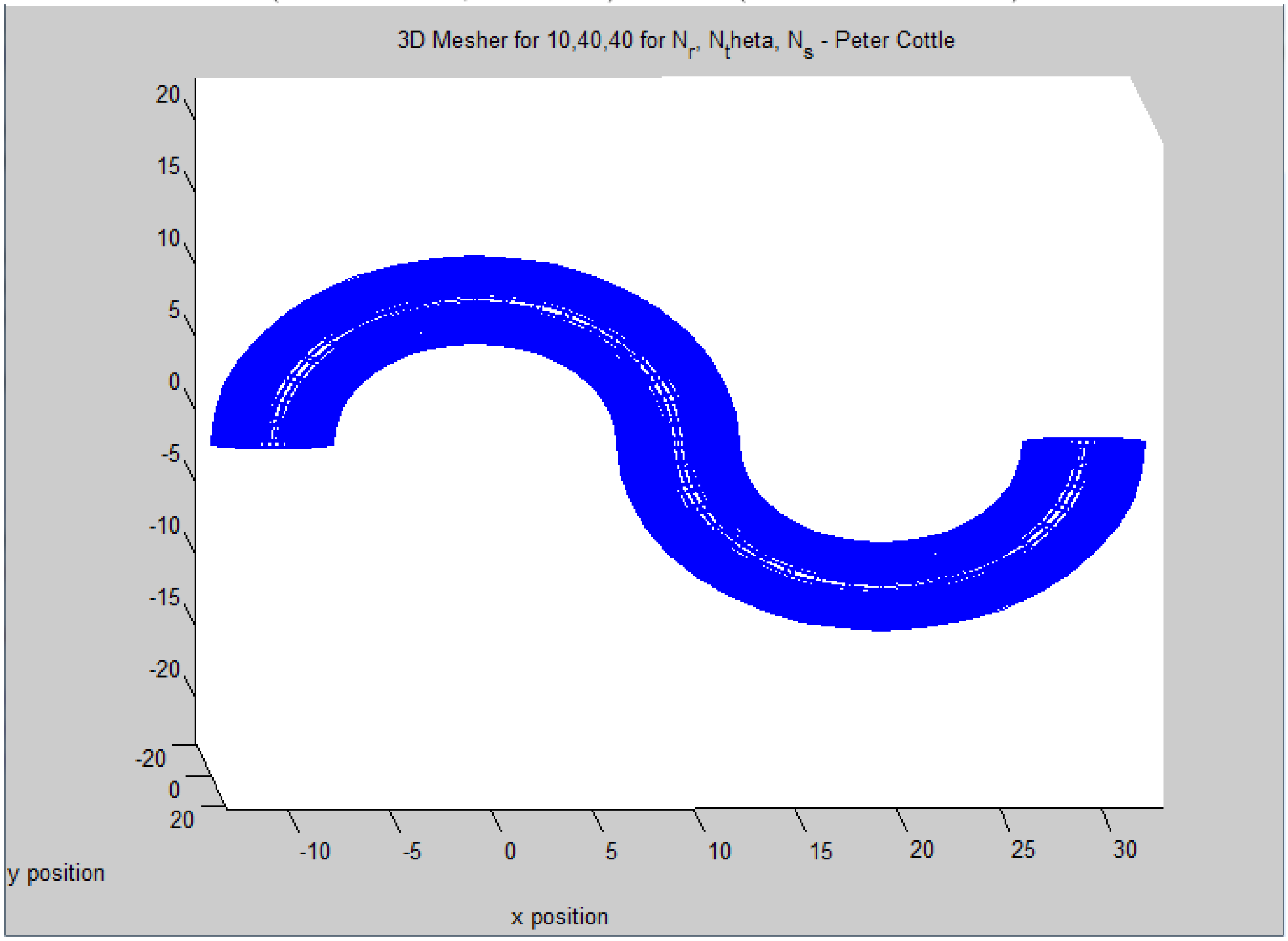
For this mesh, only element connectivity is shown, for the star plotting became visually obstructive:

$$(N_r, N_\theta, N_s) = (5, 20, 20)$$



Finally, for this plot, there are essentially **128,000** lines being drawn (for all the element connectivity) and thus the plot is difficult to see. The hollow center section can be seen though:

$$(N_r, N_\theta, N_s) = (10, 40, 40)$$



Time-Transient Case

First, we will define the time transient case for the IMPLICIT formulation. The important thing to note here is that the expression will get somewhat ugly because of the second derivative, so we will define this term:

$$\Psi \overset{\text{def}}{=}$$

$$\nabla \cdot \sigma + f$$

Then taking this equation:

$$\nabla \cdot (E : \nabla u) + \rho b = \rho \ddot{u}$$

We use a finite difference to approximation the second derivative like in **Homework 5, which I will refer to for a more rigorous derivation of the finite difference for a time derivative.** The expression then yields (after solving for u + delta t):

$$\int_{\Omega} \rho_o v \cdot u(t+\Delta t) d\Omega = \int_{\Omega} \rho_o v \cdot u(t+\Delta t) d\Omega + \int_{\Omega} \rho_o v \cdot (\frac{\partial u}{\partial t}|_{t+\Delta t})\Delta t + \int_{\Omega} v \cdot (\Delta t)^2 \Psi(t+\Delta t) d\Omega$$

For the explicit version, we simply move all the evaluated terms back one timestep to obtain a simpler equation to evaluate:

$$\int_{\Omega} \rho_o v \cdot u(t + \Delta t) d\Omega = \int_{\Omega} \rho_o v \cdot u(t) d\Omega + \int_{\Omega} \rho_o v \cdot (\frac{\partial u}{\partial t}|_{t})\Delta t + \int_{\Omega} v \cdot (\Delta t)^2 \Psi(t) d\Omega$$

For the lumped mass approximation, we will **ignore the deformation of the element** and instead use a diagonal mass matrix. This is essentially equivalent to dropping the first derivative term (I think):

$$\int_{\Omega} \rho_o v \cdot u(t+\Delta t) d\Omega = \int_{\Omega} \rho_o v \cdot u(t+\Delta t) d\Omega + \int_{\Omega} v \cdot (\Delta t)^2 \Psi(t+\Delta t) d\Omega$$

Discussion

This homework was quite length, and in order to save the reader's time and a small fraction of some tree, I will omit the typical length discussion at the end of the report. In conclusion, this homework gave me a *very* in-depth summary of the finite element formulation in three dimensions and all of the aspects to consider when choosing approximation functions, implementing the formulation, post-processing, and deriving the weak form. This will be good preparation for the final and perhaps a career in FEM software.

APPENDIX:

Like in homework #1, the raw data for the above figures is not provided here but can be provided upon request. It is also quite easily generated from the included matlab code below.

```
function [ gNodeToCoord, connectMap ]
=
archMesher(r_step,theta_step,s_step,s
houldPlot,big_radius)
r_end = 3;
r_begin = 2;
r_range = r_end-r_begin;
theta_begin = 0;
theta_end = 2*pi;
theta_range = theta_end-theta_begin;
s_begin = 0;
s_end = 2*pi;
s_range = s_end-s_begin;
numPoints = (theta_range/theta_step +
1) * (r_range/r_step + 1) *
(s_range/s_step+1);
gNodeToCoord = zeros(numPoints,3);
coordNum = 1;
if(shouldPlot)
    close all
    figure(1)
    hold on
    axis([-2*big_radius 2*big_radius
-2*big_radius 2*big_radius -
2*big_radius 2*big_radius])
end
for s=s_begin:s_step:s_end
    for
theta=theta_begin:theta_step:theta_en
d
        for r=r_begin:r_step:r_end
            %get the x
            x = r*cos(theta);
            y = r*sin(theta);

            z = 0; %z is default zero
for now, we will translate and rotate
            %later

            gNodeToCoord(coordNum,1)
= x;
            gNodeToCoord(coordNum,2)
= y;
            gNodeToCoord(coordNum,3)
= z;
            coordNum = coordNum + 1;
        end
    end
end
%ok now we have generated each slice
of the entire thing, but they are all
%still on top of each other! we need
to rotate and translate each one :P
```

```
disp 'Hi!'

shouldplot = true;

N_theta = 10;
N_r = 2;
N_s = 20;


big_radius=10;

r_step = 1/N_r;
theta_step = 2*pi/N_theta;
s_step = 2*pi/(N_s);

[gNodeToCoord, connectMap] =
archMesher(r_step,theta_step,s_step,shouldpl
ot,big_radius);

%plotting should be done...
function [ x,ys,theta ] =
getPosOnArch(big_radius,s_step,should_plot)
%GETPOSONARCH Summary of this function goes
here
%   Detailed explanation goes here
%first, if x is less than 2*big_radius...
total_length = 3*big_radius;
%cant divide up the x's like this, we need
another way
daAngle = [-pi:s_step:pi];
x = zeros(max(size(daAngle)),1);
for i=1:length(daAngle)
    thisAngle = daAngle(i);
    thisX = big_radius*cos(thisAngle);

    if(i > 1)
        if(sin(thisAngle) > 0)
            thisX = thisX + 2*big_radius;
        end
    end
    x(i) = thisX;
end
atHalf = round(length(x)/2);
x(end) = x(end) + 2*big_radius;
temp = x(atHalf+1:end-1);
temp = flipud(temp);
x(atHalf+1:end-1) = temp;
x = x';
ys = zeros(max(size(x)),1);
theta = zeros(max(size(x)),1)-100;
for i=1:length(ys)
```

```
% so do this with a rotation first
about the y axis of whatever theta we
% want, and then translate
% we need homogenous coordinates for
this too just FYI, so this might be a
% beezy but whatever
numPointsInSlice =
(theta_range/theta_step + 1) *
(r_range/r_step + 1);
%divide up each into slices
totalSlices = numPoints /
numPointsInSlice;
%ok now we have to loop through each
slice, splitting it up, rotating it,
%transforming it, and then putting
the data back in
%get all x, y, and theta's
[xs, zs, thetas] =
getPosOnArch(big_radius,s_step,false)
;
xs = xs';
for currentSlice=1:totalSlices
    %get the rotation matrix for this
part
    p = thetas(currentSlice,1);
    rotAmount = 2*pi-p;

    rotMatrix = [   cos(rotAmount)
0   sin(rotAmount)  ;
                    0
1   0               ;
                    -sin(rotAmount)
0   cos(rotAmount)  ];
    %     pause
    %also get the translation matrix
for this

    %we need to translate none in the
y, but a certain amount in
    %z and a certain amount in x

    %the radius is just big_radius.
the x amount is
    %cos(phi) * big_radius,
    %Z amount is sin(phi) *
big_radius

    xAmount = xs(currentSlice,1);
    zAmount = zs(currentSlice,1);

    tMatrix = [ 1         0          0
xAmount;
                0         1          0
0;
                0         0          1
zAmount;
```

```
    if(x(i)<=big_radius)

        center = 0;

        ys(i) = sqrt(big_radius^2-x(i).^2);

        theta(i) = atan(ys(i)./x(i));

        if(theta(i) > 0)
            theta(i) = theta(i) - pi;
        end

        if(theta(i) == 0 && x(i) > 0)
            theta(i) = -pi;
        end

    else
        %if not, subtract the second big
radius and flip it
        ys(i) = -sqrt(big_radius^2-
(big_radius*2-x(i)).^2);

        center = (big_radius*2);
        theta(i) = atan(-ys(i)./(center-
x(i)));

        if(theta(i) > 0)
            theta(i) = theta(i) - pi;
        end

    end
end
if(should_plot)
    figure(2)
    clf
    plot(x,ys)
end
end
%plot3(gNodeToCoord(:,1),gNodeToCoord(:,2),g
NodeToCoord(:,3),'b*');
edge = -big_radius*2 - r_end;
edge2 = -big_radius-r_end;
axis([edge2, -edge2+big_radius*2, edge, -
edge, edge, -edge])
pause
%now go through and do connectivity
%we need the number of elements now
strangely enough...
%num elements is the same as the number of
elements per slice
%times the number of slices. so.
numElementsPerSlice = r_range/r_step;
numSlices = theta_range/theta_step;
numElements = numElementsPerSlice*numSlices;
connectMap = zeros(numElements,8);
```

```
                 0         0           0
1;];
%     pause
    %ok so now go through and
translate each one of these coords

    startNode =
numPointsInSlice*(currentSlice-1) +
1;
    endNode =
startNode+numPointsInSlice-1;

    for i=startNode:endNode
        x = gNodeToCoord(i,1);
        y = gNodeToCoord(i,2);
        z = gNodeToCoord(i,3);

        vector = [x; y; z];

        vector = rotMatrix*vector;

        homoVec = [vector;1];

        homoVec = tMatrix*homoVec;

        newX = homoVec(1,1);
        newY = homoVec(2,1);
        newZ = homoVec(3,1);

        gNodeToCoord(i,1) = newX;
        gNodeToCoord(i,2) = newY;
        gNodeToCoord(i,3) = newZ;

    end


end
```

```
%make the connectivity map
%start with sliceCounter and sideCounter
elementCounter = 1;
%REALLY BAD VARIABLE NAMING FOLLOWS
for sCounter=1:totalSlices-1
    for sliceCounter=1:numSlices
        for
sideCounter=1:numElementsPerSlice
            %get the lower left
            LL = sideCounter +
(sliceCounter-1)*(numElementsPerSlice+1) +
(sCounter-1)*numPointsInSlice;
            LR = LL + 1;
            UL = sideCounter +
(sliceCounter)*(numElementsPerSlice+1) +
(sCounter-1)*numPointsInSlice;
            UR = UL + 1;

            LLU = LL + numPointsInSlice;
            LRU = LR + numPointsInSlice;
            ULU = UL + numPointsInSlice;
            URU = UR + numPointsInSlice;
            %plot this
            if(shouldPlot)
                xs = [gNodeToCoord(LL,1)
gNodeToCoord(LR,1) ...
                      gNodeToCoord(LR,1)
gNodeToCoord(UR,1) ...
                      gNodeToCoord(UR,1)
gNodeToCoord(UL,1) ...
                      gNodeToCoord(UL,1)
gNodeToCoord(LL,1) ...
                      gNodeToCoord(LL,1)
gNodeToCoord(LLU,1) ...
                      gNodeToCoord(LLU,1)
gNodeToCoord(LRU,1) ...
                      gNodeToCoord(LRU,1)
gNodeToCoord(LR,1) ...
                      gNodeToCoord(LR,1)
gNodeToCoord(LRU,1) ...
                      gNodeToCoord(LRU,1)
gNodeToCoord(URU,1) ...
                      gNodeToCoord(URU,1)
gNodeToCoord(UR,1) ...
                      gNodeToCoord(UR,1)
gNodeToCoord(URU,1) ...
                      gNodeToCoord(URU,1)
gNodeToCoord(ULU,1) ...
                      gNodeToCoord(ULU,1)
gNodeToCoord(UL,1) ...
                      gNodeToCoord(UL,1)
gNodeToCoord(ULU,1) ...
                      gNodeToCoord(ULU,1)
gNodeToCoord(LLU,1) ...
                       ];
                ys = [gNodeToCoord(LL,2)
```

```
gNodeToCoord(LR,2) ...
                       gNodeToCoord(LR,2)
gNodeToCoord(UR,2) ...
                       gNodeToCoord(UR,2)
gNodeToCoord(UL,2) ...
                       gNodeToCoord(UL,2)
gNodeToCoord(LL,2) ...
                       gNodeToCoord(LL,2)
gNodeToCoord(LLU,2) ...
                       gNodeToCoord(LLU,2)
gNodeToCoord(LRU,2) ...
                       gNodeToCoord(LRU,2)
gNodeToCoord(LR,2) ...
                       gNodeToCoord(LR,2)
gNodeToCoord(LRU,2) ...
                       gNodeToCoord(LRU,2)
gNodeToCoord(URU,2) ...
                       gNodeToCoord(URU,2)
gNodeToCoord(UR,2) ...
                       gNodeToCoord(UR,2)
gNodeToCoord(URU,2) ...
                       gNodeToCoord(URU,2)
gNodeToCoord(ULU,2) ...
                       gNodeToCoord(ULU,2)
gNodeToCoord(UL,2) ...
                       gNodeToCoord(UL,2)
gNodeToCoord(ULU,2) ...
                       gNodeToCoord(ULU,2)
gNodeToCoord(LLU,2) ...
                        ];


                 zs = [gNodeToCoord(LL,3)
gNodeToCoord(LR,3) ...
                       gNodeToCoord(LR,3)
gNodeToCoord(UR,3) ...
                       gNodeToCoord(UR,3)
gNodeToCoord(UL,3) ...
                       gNodeToCoord(UL,3)
gNodeToCoord(LL,3) ...
                       gNodeToCoord(LL,3)
gNodeToCoord(LLU,3) ...
                       gNodeToCoord(LLU,3)
gNodeToCoord(LRU,3) ...
                       gNodeToCoord(LRU,3)
gNodeToCoord(LR,3) ...
                       gNodeToCoord(LR,3)
gNodeToCoord(LRU,3) ...
                       gNodeToCoord(LRU,3)
gNodeToCoord(URU,3) ...
                       gNodeToCoord(URU,3)
gNodeToCoord(UR,3) ...
                       gNodeToCoord(UR,3)
gNodeToCoord(URU,3) ...
                       gNodeToCoord(URU,3)
gNodeToCoord(ULU,3) ...
```

```
                      gNodeToCoord(ULU,3)
gNodeToCoord(UL,3) ...
                      gNodeToCoord(UL,3)
gNodeToCoord(ULU,3) ...
                      gNodeToCoord(ULU,3)
gNodeToCoord(LLU,3) ...
                       ];
                line(xs,ys,zs);
                hold on;
                %axis([-3 3 -3 3]);
                %pause(0.05*sCounter);
                %pause(0.01);
            end
            connectMap(elementCounter,:) =
[LL LR UR UL LLU LRU URU ULU];
            elementCounter = elementCounter
+ 1;
        end
    end
end
end
```