

Assessment of Radius Bone Strength in Torsion with the Finite Element Method – Report and Documentation

Daniel Baker

Schulich School of Engineering

Supervisor: Dr. SK Boyd

December 12th, 2008

Introduction

The purpose of this research was to test FE models of the distal radius against actual mechanical loading. During the summer of 2007 ten cadaveric radii from 5 donors were tested using an Image Guided Failure Analysis system. Of each pair one radii was tested in compression and the other in torsion. The failure load determined from compression testing was used to load the other half of the pair to 50% of failure before being rotated to torsion failure.

Prior to this project the in-house finite element meshing software did not have functionality to simulate torsion. The program used is called AppDirectMechanicsGenerator, and it calculates an input file describing a finite element mesh. This mesh is then solved using the solver parafaim. Various boundary conditions and nodal displacements could be set for compressive or shear loading, but there was no way to calculate or impose the displacements caused by rotation. The first stage of the project was therefore coding new mesh writer functionality.

VTK

The in-house meshing software is built using vtk or the Visualization Tool-Kit. Vtk is a C++ library of objects and classes that processes image data into usable forms. Software that uses vtk is design using a basic structure called the visualization pipeline where the output of one class is the input for the next.



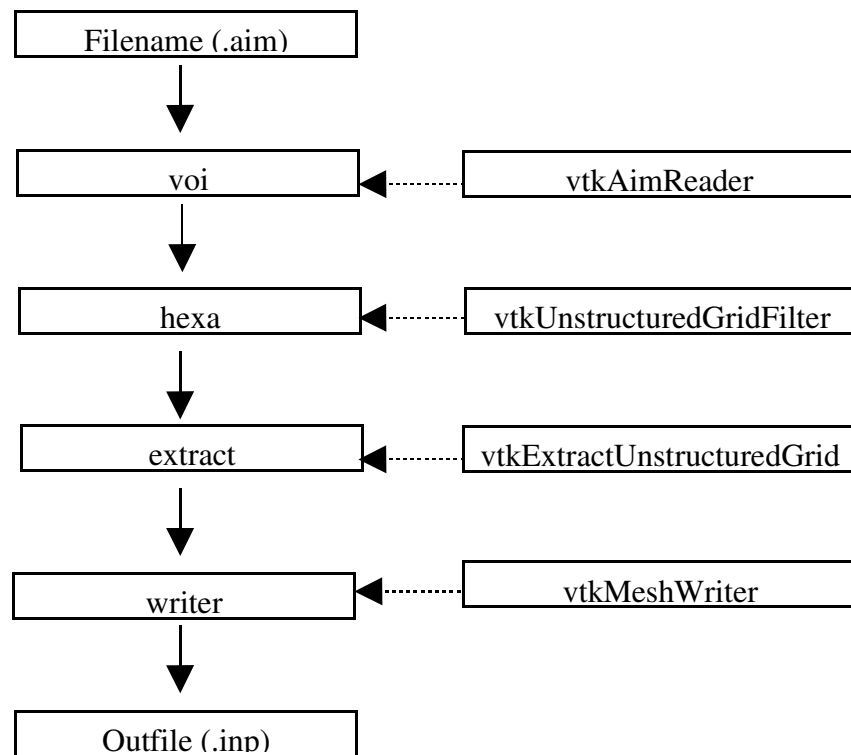
You pass input data to class A, and ask D to provide the output. D asks C to update, so C asks B to update, so B asks A to update. A uses the input data and executes its function, passes the output to B and asks it to execute, B passes its output to C and asks it to execute, C passes its output to D and asks it to execute, D provides the result. The back propagation of updates and forward propagation of executes ensures that the newest results are used throughout the process.

AppDirectMechanicsGenerator

(On Douglas: ~/pack/vtkMy/src/Applications/PureVTK/ DirectMechanicsGenerator/
DirectMechanicsGenerator.cxx)

This is the actual C++ program that executes the various vtk functions and classes to create finite element meshes. One of the major classes used, vtkMeshWriter, is in-house code and will be described in full later.

The basic structure of execution upon which DMG is based is as follows:



This structure can be seen in the DMG source code on lines 232 to 275. An object of each class type is created and its input is set to the output of the previous object.

Of interest is the functions used to set the value of variables. The functions Set and Get are automatically created by macros at compile and are not explicitly defined in the source code. Their function is controlled by overloaded function declarations in the header files (.h). For example, in vtkMeshWriter.h lines 196 to 430 tell the compiler how to handle each use of Get and Set based on which variable name is used. Lines 418 to 430 contain the new variable Get and Set instructions added for torsion.

The rest of the program send vtkMeshWriter different instructions based on what kind of finite element test has been requested. Lines 304 to 328 contain the new instructions added for the torsion case. Once these have been executed the program exits.

We will now look at the functionality added to vtkMeshWriter.

vtkMeshWriter

(On Douglas: ~/pack/vtkMy/src/Unsorted/vtkMeshWriter.cxx, and vtkMeshWriter.h)

The purpose of this class is to calculate the actual displacements of each node directly affected by the imposed test, and write a output file describing the locations of all nodes, the elements each node belongs to, and the constraints on the nodes (including displacements and boundary conditions).

A point of interest: Lines 223 to 242 -> this defines model faces is in terms of the x, y, and z directions. Face00 and face01 are the top and bottom surfaces (horizontal) and face10/11 and face20/21 are the vertical faces. Their value is set the the minimum or maximum in that direction. For example, if the model is a 10x10 cube faces 00, 10, and 20 will all equal 0 and 01, 11, and 21 will equal 10. From this point on the faceXX variables will be used to automatically tell a function where to look for a face. This is also explained at the very end of the file.

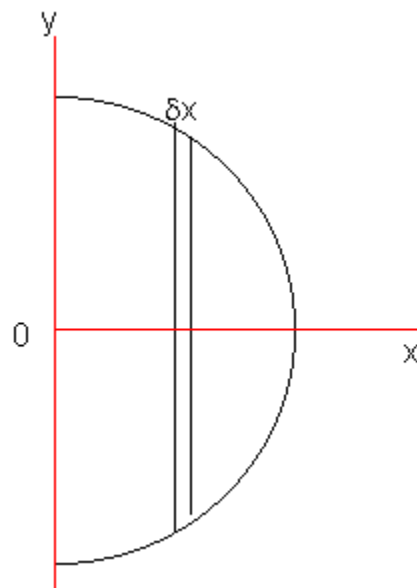
The most significant part of the new code is found from line 697 to 811. Most of its function is explained in the code comments, but there are a few points that could use more elaboration.

Centroid Calculation (lines 726 to 768)

This section uses a center of mass style calculation to determine where the center of the top surface is. The basic formula for the centroid in one axis is given by:

$$C_x = \frac{\sum x \, dA}{\sum dA}$$

In theory the mass is split into infinitesimal slices of width dx , but in our case the slice thickness is already known from the finite element dimensions. The area of the slice dA is found as the product of slice thickness and height. The sum of all slice areas times their position is divided by the total area to give an average position. The procedure is then repeated in the other dimension to find the 2D centre of mass.



The code starts in the offAxis1 direction (default is x,

corresponding to an onAxis direction of z). iterating through the width of the model. The for loop at line 731 goes through all the points in the model and determines whether they are in the slice of interest. Any point that are have their offAxis2 dimension compared to a max and min, replacing them if it is greater or less. In this way the upper and lower extent of each slice is found. The area is calculated and added to a running total. Once the entire width has been examined the total weighted area is divided by the total area to find the centroid. The code then repeats the process in the offAxis2 direction.

Note: This is very slow. It could be sped up by looking at only the top face nodes. This would be a simple matter of figuring out how to iterate through the list faceTopNodeIds and finding the node position using input->GetPoint.

Immediately following this code (lines 770 to 775) is the manual center of rotation functionality. To use this functionality the use input three numbers. The first is a flag (0 or 1) and the other two are dimensions. The flag 0 indicates that the user simply wants to shift the center slightly. The center of rotation is calculated as described above, and then moved according to the supplied dimensions. The flag 1 indicates that the user want to set the center themselves. The calculation is skipped (line 724), the center is set to zero, and then is moved according to the supplied dimensions.

Nodal Displacements (line 781 to 806)

The next step is to use the angle of rotation to determine how much each top face node will have to be displaced. This code uses the coordinate rotation method. Given the node's position and the centroid we can find each node's position in polar coordinates (described in terms of an angle and radius)

$$x = R \cdot \cos(\alpha) \quad y = R \cdot \sin(\alpha)$$

We can then rotate the point around the center by the specified amount θ .

$$x' = R \cdot \cos(\alpha + \theta) \quad y' = R \cdot \sin(\alpha + \theta)$$

By using the trigonometric rules

$$\cos(\alpha + \theta) = \cos(\alpha) \cdot \cos(\theta) - \sin(\alpha) \cdot \sin(\theta) \quad \text{and} \quad \sin(\alpha + \theta) = \sin(\alpha) \cdot \cos(\theta) + \cos(\alpha) \cdot \sin(\theta)$$

we then get

$$x' = R \cos(\alpha) \cdot \cos(\theta) - R \sin(\alpha) \cdot \sin(\theta) \quad \text{and} \quad y' = R \sin(\alpha) \cdot \cos(\theta) + R \cos(\alpha) \cdot \sin(\theta).$$

The combinations of R with $\cos(\alpha)$ and $\sin(\alpha)$ are of course the original values of x and y. This finally gives us our new node positions:

$$x' = x \cdot \cos(\theta) + y \cdot \sin(\theta) \quad y' = y \cdot \cos(\theta) - x \cdot \sin(\theta) \quad (\text{lines 798-799})$$

The nodal displacement vector is simply the difference between the new and the original positions:

$$(x' - x, y' - y) \text{ (lines 803-804)}$$

The code on lines 777-779 allow the user to check what the center of rotation was defined as, and that the program has correctly identified the top surface.

Note: a useful check would be to flip a model and run the same analysis with the rotation displacements being imposed on the opposite end. The results should be the same either way.

The file print coded on lines 789, 801, and 809 is used to create a top face node list for use later. It will be needed when we calculate total imposed torque using a separate program.

Up to this point we have been in the main function of vtkMeshWriter, WriteData(). This function does all the calculation, and then calls another function to write the output file. Line 899 calls the function WriteFaimMeshOutput, which takes us to line 1738. This function prints all the necessary node data to a file, and then on line 2052 calls another function to write the actual node displacement constraints. A if statement distinguishes between that standard compression and torsion cases. For torsion we call the function WriteFaimTorsionData and go to line 2146. Here the lists of constraints are used to write out the amount of displacement for each relevant node. First the top face is compressed (the torsion code would in sync with compression), then displaced in the two off axis directions. Each line in the output file contains the node ID, then the direction, then the amount of displacement.

A possible point of confusion is the constant use of +1 throughout the code. These are used to translate between a 0-based and 1-based counting system. vtkMeshWriter uses a 0, 1, and 2 to describe z, x, y, (for the default test orientation) but the FE code uses 1, 2, 3. Also, in the constraint lists the first position in each row contains the node ID, so the on axis displacement is actually in the onAxis+1 (or 1) position.

Compute_Torque

(On Douglas: ~/dcbaker/fem/compute_torque.py)

Once the model has been solved, the apparent level torque has to be found for comparison to the mechanical testing data. This is functionality that could be built into Parafaim at some point.

This program takes the list of top node ID's and postions, the .dat file produced by FE analysis, and the two centroid positions in offAxis1 and offAxis2 directions (default x, y). It first reads in all the top face node positions (line 21) then opens the .dat and scans until it find the nodal forces header. It then reads in all the reaction forces. The torque is found from the sum of forces in the x-direction acting at a distance of y from the center plus all the forces in the y-direction acting at a distance of x from the center.

$$T = \sum \left[\left(x_i - C_x \right) \cdot R_{yi} + \left(y_i - C_y \right) \cdot R_{xi} \right]$$

Note: This software only works when the testing direction is z. The node position list (printed to nodedata.txt) is correct for any testing direction, but the nodal forces are not listed in a different order based on test direction. The indices in the code on lines 62 and 63 will have to be changed manually. If testing direction is x change to 1 and 2, and if it is y change to 0 and 2.

Using the Software – Procedure

1. In order to maximize the area that was constrained, the very top of the models were removed to give a flat surface at the widest point. The below steps can be used to select any segment of a model to be tested. Whereever an option isn't referred to, leave it as the default.
 - a) On the VMS system navigate to your project models folder
 - b) Use the command 'sap' to start SAP and select your model from the window
 - c) Use the OV Shrink to look at as many slices as possible (8 x)
 - d) Determine the number of the slice you would like to cut at and exit SAP
 - e) Open IPL using the command 'ipl'
 - f) Use the command 'sub_get'
 - g) Enter your filename, then the output filename
 - h) Now enter the subsection starting point. If you are only cutting to shorten the z dimension use '-1 -1 #'. -1 tells it to use the default x and y points, and # is the slice you would like to start your subsection at. The slices count up from slice 1 at the top of your model, so if you want to remove the top 50 slices use that as the starting point, e.g. '-1 -1 50'
 - i) Enter the dimensions of your subsection. If you are only taking the top off your model this will be the total model dimension less the slices you want removed. So if your model has 400 slices you would enter '-1 -1 350'
 - j) Don't exit IPL! Your output file doesn't exist outside of IPL.
 - k) Use the command 'write'
 - l) Enter the output filename you used for sub-get and the output filename you would like to use
 - m) Exit IPL using 'exit'
2. The aim was then transferred to Douglas for FE analysis
3. The file 'IFGA Data.xls' was used to determine the pre-torsion compression load that was used
4. The model was meshed using AppDirectMechanicsGenerator with a modulus of 6829 and 1% compression
5. The model was solved on Matrix (a Westgrid server) and the total reaction force determined (listed at the end of the .dat as Rfz)
6. The apparent level force and deformation are proportional for a linear model, so simply scale the compression so as to give the correct compressive force.
7. The model was then meshed again with DMG with the same modulus, the determined compression,

5° rotation, and a center of rotation determined by eye-balling the center of the diaphysis in the program 'vis'.

8. The models were again solved on Matrix and the output file .dat transferred back to Douglas.
9. The apparent level torque was found using compute_torque.py

Running Large Files on Matrix

The models used for this study are quite large (~500 slices, about 27000 mm³ at 82 µm resolution for a total of ~12 000 000 nodes and 9 000 000 elements) and are therefore difficult to solve. The solution is to increase the number of 'nodes' and the amount of memory you request. Matrix allows you to use 1 GB of memory per processor and organizes itself into nodes of 2 processors. In order to run the large models described before it was necessary to request 14 nodes and 28 000 MB of memory. These values are set in the .pbs file that you use to submit jobs on matrix. Westgrid recommends using MB in your request instead of GB to ensure there is no confusion.

Results

The five radii mechanically tested in torsion were modeled and solved by finite element analysis. The sample numbers are 01, 04, 05, 08, and 09. The mechanical testing scans can be found on VMS under Projects/IGFA/Models. The aim's used to develop the FE models are the NOLOAD scans which show most of the distal region of the radius including a significant portion of the diaphysis. These correspond to the actual bone sections that were tested.

<i>Sample</i>	<i>Filename</i>	<i>Compression</i>	<i>Rotation [°]</i>	<i>Modulus [MPa]</i>	<i>Centroid (x,y) [mm]</i>
1	igfa_distfore_01_noload_seg_sub_t	0.3488	5	6829	(13.429, 17.303)
4	igfa_distfore_04_noload_seg_sub_t	0.3338	5	6829	(12.611, 16.792)
5	igfa_distfore_05_noload_seg_sub_t	0.1934	5	6829	(17.693, 18.032)
8	igfa_distfore_08_noload_seg_sub_t	0.3505	5	6829	(14.485, 20.562)
9	igfa_distfore_09_noload_seg_sub_t	0.2335	5	6829	(15.726, 19.047)

These values can be found in the file 'mesh.com' on Douglas at ~/pack/vtkMy/src/bin

The apparent level torque required to rotate each model by 5 degrees is as follows:

<i>Sample</i>	<i>Torque [Nm]</i>
1	23.2119
4	4.2023
5	9.34064
8	10.2349
9	5.75210

All the input files (.inp), submit files (.pbs), and results (.dat) can be found on the Matrix server in / scratch/gmcampbe. Please contact Graeme Campbell for log-in information.

All _t files are torsion tests, all _z files are initial 1% compression tests, and all _z1 are pre-compression matched tests to confirm same loading force (except sample 01, which is called z6).