Avalon

2D and 3D Truss FEA

Santillo, Shawn

M168

Getting Started

Upon starting the program, you will see that the process file and display element buttons are greyed out. This is because you must first load your elements. To do this, click the load button and select your inp file. Upon loading, the process and display buttons will become active. Note that some variables (such as the global stiffness matrix) are created in processing – thus, if you display elements without processing first, these elements will not be displayed. Once the processing is complete, the displacement matrix D will be displayed in the console. Note that you can select and copy the text in this console window – this is by design, in case you wish to copy parts of it for any reason.

If your inp file has errors, you will receive an error message describing the problem. The line which caused the problem will also be output, as well as the line number. This allows for extremely easy debugging. A list of the errors with respect to the input file is shown below:

- "Error: TYPE= or ELSET not found in line " + std::to_string(lineNumber) + ". Line shown below:\n" + line;
- "Error: Duplicate element name found in line " + std::to_string(lineNumber) + ". Line shown below:\n" + line;
- "Error: Element of type T2D2 declared after 3D nodes declared. Cannot mix dimensions (sounds cool though). See line " + std::to_string(lineNumber) + ". Line shown below:\n" + line;
- "Error: Element of type T3D2 declared after only 2D nodes declared. Cannot mix dimensions (sounds cool though). See line " + std::to_string(lineNumber) + ". Line shown below:\n" + line;
- "Error: Section not defined before moving to post section information at line number: " + std::to_string(lineNumber)+ ".\n"+line)
- "Error: Element name from line " + std::to_string(lineNumber) + " not found. Line shown below:\n" + line + "\nNon existant element name from line was: " + split[1] + ".");
- "Error: Duplicate material name found in line number " + std::to_string(lineNumber) + ". Line shown below:\n" + line);
- "Error: Section not defined before moving to post section information at line number: " + std::to_string(lineNumber)+ ".\n"+line);
- "Error: Tried to enter element nodes but no element exists yet. Line number: " + std::to_string(lineNumber)+".\n"+line);
- "Error: You are trying to assign a node which does not exist yet to an element. Line number: " + std::to_string(lineNumber)+".\n"+line);
- "Error: The area in line number: " + std::to_string(lineNumber)+" is not a correctly formatted double.\n"+line);

In the case that your inp file is empty or you do not exit the file selection window without selecting a file, current variables will be wiped and the program will require a valid file to be correctly submitted to prevent errors.

Code Design

How it works / Pseudocode

This pseudocode section will be divided into three sections – loading the file and the preprocessing, the processing which occurs when the process file button has been clicked, and the display elements button.

How it works / Pseudocode – The Load Button

We begin our journey in the following function:

void MainWindow::on_loadFileButton_clicked()

We begin by calling two cleaning functions in order to wipe our global variables. These functions clear the global variables that we are using to hold and transfer data between functions. This is called such that we do not append input information to preexisting variables.

Next, we prompt the user to load their input file, and begin to read the selected input file line by line with a while loop. For each line in the while loop we go through the following routine:

- We erase all preceding white spaces from the line.
- We skip the line if it is empty or begins with "**" (meaning that it is commented out).
- We have confirmed that our current line is valid input. We now have an if statement, where we check if the first character of the line is '*'. If it is, it means that we are entering a sectional level. This routine is as follows:
    - We figure out what section that we should be entering. Chained if statements check the beginning of the line to see what subroutine should be entered (such as *NODE and *ELEMENT). We then set the section variable to indicate what section we are currently in for subsequent lines without '*'. If there is no processing to be done on this line, we continue (such as in the case of *NODE).
    - Once we have hit a positive on an if statement, we parse the line into a string vector delimited by white spaces using std::stringstream. We then sanitize our resultant vector, removing the orphan whitespaces, commas and parts of the strings when relevant. We then read our sanitized and delimited values into a temp of the correct type of variable, and push this into the correct global storage vector.
- We now handle the case of the above if statement evaluating to false:
    - In this case, the line does not start with the character '*'. We exit with an error if the section has not been defined yet (ie a preceding line was not called with the '*' as the first character such as in *NODE).
    - In the case that the input is correct, we begin by using the line to create a string stream which we then split and push into a string vector based on delimiting by whitespaces. We then cycle through and sanitize the string vector, removing commas and whitespaces.
    - Remember that we are in a subroutine based on the section value, which was set in the block of code handling lines beginning with the '*' character. Using this section variable, we have entered the correct block / subroutine for the data that is contained in our current line, which has already been sanitized. We push and/or apply (depending on the section) the data read from the line into our global vectors and variables. Note that

the data is generally converted to its respective correct type (when said type is not an std::string, which it is already in) using the std::sto_ function series.

- We have now cycled through the entire input file and read the data. We output to console that the file loaded with no errors. We set the disabled (greyed out) buttons to an enabled state, and update the gui that no process is running.

How it works / Pseudocode – The Process Button

We are now able to process our loaded data (when nothing is loaded, the button is greyed out).

```
void MainWindow::on_processButton_clicked()
```

We begin by assigning the code numbers and creating the force vectors for our elements using the following function:

```
arma::mat assignCodeNumbersAndCreateForceVector(std::vector<element> &elements,
                                                std::vector<node> &nodes)
```

This function takes our global vectors of elements and global vector of nodes as inputs passed by reference as arguments and returns a matrix of type arma::mat, which is the matrix of the armadillo library. Note that an overview on struct design and members is available in the variable structures section of this document.

Note that to avoid creating duplicate functionality for different dimensions using different methods, we treat 2D nodes as being in 3D and all sharing the same z coordinate (0) and being constrained with respect to the z dimension.

We will now cycle through the global nodes vector, which contains all of the nodes from the input file.

- For each node, we check the degree of freedom constraints and set the boolean values for each direction (x, y, and z) to true or false based on the degree of freedom constraints.
- We now initialize a ticker which will increase by one each time we assign a code number to a direction.
- Now for the current node for each of the x, y, and z directions:
    - We check if the direction is constrained. If it is, we exit this subroutine.
    - If it is not constrained, we set the code number of the current node in the current direction to the ticker (which as mentioned before is now incremented).
    - We also calculate the force on this node in the current direction, and add this to our force vector.
- After this for loop completes, all non-constrained directions will have been assigned a code number and the force vector will have been created.
- We now cycle through the elements global vector and set the sub member nodes to have the same code values as in our standalone global node vector.
- The global vector *elements* has now been updated correctly with the values from the global vector *nodes*.

- We now convert the force vector to an arma::mat, and return that value (as the return type of the function requires).

We have now successfully created the force vector in the correct format. We output the created force vector to the console. In this same step we have also created and applied the code numbers for each 3D direction (remember that we are computing 2D in 3D with z always equal to zero and the z direction always constrained).

We now call the member stiffness matrix function:

```
void createMemberStiffnessMatrices(std::vector<element> &elements)
```

We begin by looping through the passed elements vector in a for loop. For each element, we then loop through subelements. It is at this level which the member stiffness matrices exist, and thus the level they are created at. We now follow this routine:

- We begin by calculating the cosine angles for each direction. These are stored in cx, cy, and cz.
- We create the matrix T (of type arma::mat) out of the cosine angles:
- $T = \begin{bmatrix} cx & cy & cz & 0 & 0 & 0 \\ 0 & 0 & 0 & cx & cy & cz \end{bmatrix}$
- We create a second matrix, kp:
- $kp = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$
- We now calculate the member stiffness matrix K as:
- $K_{member} = \frac{EA}{l} * T' * kp * T$
- Where T' is the transpose of T. We then apply this to the current subelement variable, and continue to the next instance of the for loop.

Once we have cycled through all of the elements and all of each element's subelements, we have finished creating the member stiffness matrices.

We now create the global stiffness matrix K, with the following function:

```
arma::mat createGlobalStiffnessMatrix(std::vector<element> &elements, const std::vector<node>
nodes, int stiffnessSize)
```

Note that we have an input called stiffness size here, of type int. The reason for this is that unlike matlab, memory preallocation for our matrices is required – thus, we must know the size of our matrix in advance. This found from the size of the force matrix, which was created earlier.

Once again, we cycle through each element, and each of said element's subelements using a double for loop.

- In class, we marked the code numbers of each subelement on the boundaries of the member stiffness matrix to assemble the global stiffness matrix. The code does this by creating a 3D matrix of size [6][6][2] called codeMap.
- In this codeMap matrix we fill the vertical and horizontal codes at each point in the member stiffness matrix. Note that 3D means the first two dimensions are [6], and the 3rd dimension is [2] to account for the vertical and horizontal codes.

- With the codeMap correctly populated, we move to applying the current subelement to the member matrix.
    - We cycle through all members of the subelement's member stiffness matrix using another double for loop.
    - Note that the codeMap matrix can essentially be overlaid on the member stiffness matrix (both have first two dimensions of [6][6]).
    - We now use the codeMap to apply the member stiffness matrix to the global stiffness matrix at the correct position. If the codeMap indicates that this position on the member stiffness matrix is constrained, we move to the next iteration of the loop.

We now return the global stiffness matrix, and print it to console.

We now have both the forces and the global stiffness matrix (K and Q). We now solve for the displacements:

```
arma::mat Qprime = trans(Q);

arma::mat D = eigenArmaSolve(K, Qprime);
```

We have now found the displacement matrix D, and print it to console. File processing has now completed.

How it works / Pseudocode – The Display Button

This is a straightforward section, focusing on what happens when you click the display button.

```
void MainWindow::on_displayElementsButton_clicked()
```

If processing on the current data has not been completed yet, this function will cycle through and print all variables that are created in the scope of the load file button. If the data has been processed, additional variables such as the code numbers which are only created after the process function has been called will be displayed.

Dependancies

These computations required the use of linear algebra, which is not supported by default c++. Thus, this implementation uses the armadillo and eigen libraries (both available as pure c++ source). Note that armadillo is normally installed with other dlls such as openBLAS to extend its functionality. However, I compiled this in Qt Mingw32, and the only openBLAS windows binaries I could find were for the vs compiler. Thus, to get around the problem of missing compatibility I wrote some functions to convert from matrices of the armadillo library (arma::mat) to the eigen matrices (Eigen::Matrix). Because the eigen library was written by heathens who capitalize their structs, arma::mat is mostly used, and when a function is needed from the eigen library we simply convert, call the function, and convert back to arma::mat immediately.

The gui was created using Qt, meaning that it requires Qt dlls to run. These are dynamically linked as statically linking them in the exe was making the exe have compatibility issues between different versions of windows.

The .exe is currently windows only, but a mac equivalent should be easy to produce as Qt is cross-platform and I do not call any windows specific functions.

Extendability

The single greatest benefit of writing my own parser and having everything in structs (ie object oriented) is that unlike in matlab, extendibility is simple. Almost all submembers are described not by a single type, but by an std::vector<type>. This allows for easy additions of additional members. This is best seen in that our subelements currently only have two nodes associated with each, but can be easily extended to have x nodes due to the usage of std::vector. This is noted more in our discussion of the subelement class in the structures section of this document. This design is also seen through the use of string streams to read the file – for those of a matlab background, this could be described as a dynamic form of dlmread, but much more powerful: due to the two layer usage of std::vector (in both the string stream itself and in the string of the string stream) we are able to manipulate and read lines of any length and design and apply their values to the structs without the issues of things like byte padding that you may experience in matlab when not using cell arrays.

The other large benefit of this is that unlike a large script, the functions are compartmentalized. This will allow us to extend for the final project when slightly different methods of computations are used, as we can swap out the parts that need to be modified to be ran on a subelement.type basis, and avoid having to repeat work.

Difficulties associated with the C++ style design

There are two major difficulties, though one is basically no longer a problem now. The first was how long this initially took, as basically everything had to be created from scratch except for the matrix libraries (discussed in dependencies section). However, this is now fine as since the core has been created, extension is easy (discussed in extendibility). The second problem (which may never be a problem, but we will see once the final project specs have been released) will be symbolic integration and derivatives. I currently have no support for this built. I see that there is a library for this, "SymbolicC++", which shall hopefully solve this issue if it comes up.

Misc

Note that the mainwindow.cpp in particular has a high density of comments, so if you are interested in the code directly it is good to look at the load and process functions and follow the comments through.

Variable Structures ("elements.h")

In this section we cover the three most important structs – element, subelement, and node.

element

```
struct element{
    //Abaqus allows an element to have subelements such as seen in the
following clip:
    /*
    *ELEMENT, TYPE=T2D2, ELSET=FRAME
    * **Subelement name, node number, node number....
    11, 101, 102
    12, 102, 103
    */
    std::vector<subelement> subelements;
    std::string type;
    std::string name;
    std::string materialString;
    std::string crossSectionType;
    long double area;
    //Because the material is defined after the material name of an element
is given, we will store the material name first, then as a material is
defined
    //We wills search the elements vector and assign its values to the proper
elements[i].mat members.
    //Applied on element not subelement level.
    material mat;


    /*Removed Features, keeping for reference
    //We can have any number of nodes attached to an element.
    //std::vector<node> nodes;
    //Note: Moved to subelement.
    */
};

#endif // ELEMENTS_H
```

The struct "element" is the most top level struct. When we read from the input file, we are creating an std::vector of type element. Note that an element does not have the nodes entered directly – instead, these are from its subelement member. This is how it is structured in abaqus also (as seen in our in class examples with elset=all and such).

subelement

```
//A substruct of element, as one element can have different subelements
related to nodes.
struct subelement{
    std::string name;
    std::vector<node> nodes;
    //Returns the length between the two nodes.
    long double length();

    //The basic forces vector.
    arma::mat basicForces;

    arma::mat displacementMatrix;

    //The direction cosines used for extension to 3d.
    //Making it a return function not a storage for portability and as
increased processing time is small (two operations, - and /).
    long double cx();
    long double cy();
    long double cz();

    arma::mat T;

    //The member matrix K is a member of subelement.  Relies on armadillo
library.
    arma::mat K;

    //Repeatedly computing

    /*
     In future may have to make extendable subelement class.  Reminder /
note:
     std::vector<subelement> subelements;
     */
};
```

This is the subelement class. Note that we have matrices of type arma::mat T and K – these allow us to store the cosine angle matrix and the member stiffness matrix for each subelement. Each element has a member vector of type subelement. Note that this allows us to have multiple subelements for each element.

node

```
struct node{
    //The node identifier.
    std::string name;
    //The code numbers for each of the respective directions in the node.
    int codeX, codeY, codeZ;
    //Coordinates of the node.
    long double x, y, z;
    //Constructor.
    node(long double x, long double y, long double z);
    //Default constructor
    node();
    //Positive x y and z displacements.
    long double displacements[3];
    //DOF constraints.  Needs to be initalised.
    std::vector<degreeOfFreedom>degreesOfFreedom;

    //A vector for the forces on the node.
    std::vector<force> forces;

    //Prints the debug info to qDebug().
    void printDebugInfo();
};
```

The struct node is the final struct that we shall cover. Each subelement has a vector for type node. Note that while we currently are only working with T2D2 and T2D3 subelements, having vector easily allows for T3+ types with no extra work (simply pushing back more nodes onto the subelement.nodes vector. Note that the node struct also contains code numbers of type int for x, y, and z (when applicable). These code numbers are not created for constrained dimensions. This is also the trick in how Avalon supports both T2D2 and T2D3 with the same functions – we simply render T2D2 elements in 3d by setting all z coordinates to zero and constraining nodes in the z direction (meaning that we compute 2D problems using the 3d cosine matrices, not the normal 2d matrices that we did in class). This is also extremely useful as these constraints affect the assignment of code numbers and thus reduce the returned displacement matrix to not include the 3d elements that we created for the purpose of computations.