Structural Analysis & Design

Peter Bean
SUNY Potsdam
Potsdam, New York
bean19@potsdam.edu

Timothy Mann
SUNY Potsdam
2196 Barrington Dr.
mann23@potsdam.edu

1. INTRODUCTION

SAD(Structural Analysis & Design) is an application implementing an element stiffness method solution to the analysis and design of two dimensional engineering problems. While implementing one and two degree of freedom elements, the application will serve as a framework for other engineers to implement modules for other design types. Our goal is to develop a set of engineering tools targeted to the professional engineer working on small to mid-sized projects that do not warrant the use of expensive and complicated general purpose analysis packages typical of large or complex projects.

The goal of structural analysis is to determine whether a structure is safe before it is even constructed. If a structure is unsafe then it is costly to repair and could result in injury. If the analysis is overly cautious then too many building supplies are used waisting money. Structural analysis must be safe and accurate for the results of be useful.

2. APPROACH

Our initial approach was to divide the software into two pieces and each group member would tackle his problem individually. There are a number of reasons why this did not work. The most obvious is that we had differing views on how the two components would be merged. Another problem was that only one member in the group had experience solving structural for displacements using the matrix stiffness method.

When it was realized that our divide and conquer strategy was not working we attempted to revise our design document to encompass more details. The plan was to try to flesh out as much of the application as possible in programming language agnostic form. This process continues to this day. We recognize now that we are only in the experimental stages of design. It takes a great deal of time to figure out how each component should interact with others, how to break down each component into subtasks, and how to anticipate future extensions.

Looking back it would have been wise to spend time researching other similar structural software. Unfortunately, we do not have access to any software that is suited for the audience we are writing for.

Surprisingly, there is a limited number of resources written in the last ten years on the subject of the matrix stiffness method. Most of the information available to us as students is on the Internet. The Potsdam library does have some resources, but they are older.

3. DESIGN

Java was selected as our implementation language for a number of reasons; ease of use, object oriented design, familiarity, the collections API, availability of linear algebra libraries, and compatibility with the NetBeans Platform. It was recognized early on that a purely interpreted language might not be fast enough to perform the required matrix manipulations.

The initial design is split into two parts. There is a user interface and a solver. The two parts communicate through a shared data structure. Both the user interface and the solver run in concurrent threads. The user interface is registered as a listener to the solver and the solver is registered as a listener to the user interface.

The user interface is responsible for presenting the user with an interface that is easy to understand. When the user enters a problem description into the interface the UI (user interface) is responsible for validating the users input. The UI notifies the solver when a complete problem has been specified by the user.

The solver takes validated raw user data from the shared data structure and constructs a problem representation that it is able to solve. Once the problem has been solved it returns output data to the shared data structure and notifies the UI that it is available.

3.1 Shared Data Structure

The shared data structure is a go between for the the data from the user to the solver and the solver back to the user. All operations on the structure are synchronized to avoid solving on partial data and prevent the user from receiving partial data.

This is a list of method names and their descriptions which will be available in the shared data structure.

- **getInstance** - Returns an object of class DesignBeam, DesignBar, DesignTruss, DesignFrame. Currently defaults to DesignBeam.
- **description** - A description of the problem in human readable text.
- **designCode** - Contains a string that is an index into another data structure for lookup. The lookup will return an object which contains two lists. One list will be a list of strings which are the load cases. It will be a space delimited list of symbols. The second list will be of strings which represent formulas for calculating the load.
- **designID** - A string that references the ID of the XML file.
- **designNotes** - More free text giving information about the problem.
- **forceUnit** - String that dictates the force units (lb, kips, N, kN) for the overall design.
- **propertiesForceUnit** - String that dictates force units (lb, kips, N, kN) for the material and section properties only. For example: the E and I values.
- **lengthUnit** - String that dictates the length units (in, ft, mm, cm, m) for the overall geometry.
- **propertiesLengthUnit** - String that dictates the length units (in, ft, mm, cm, m) for the elements.
- **tempUnit** - String which returns "f" for Fahrenheit and "c" for Celsius.
- **tempNonLinearity** - String specifying temperature change in individual elements. This will most likely be a null string, but could be used. For example : all=value (or member=value).

This section describes methods from a beam design (class: DesignBeam).

- **length** - get length values as a string in the current units. If called without an argument it returns a string which is guaranteed to parse as a double. The length value is in terms of the length unit type. Example : "24.124".

  If an argument is specified that sets the length of the design (one beam).
- **supports** - A string of the form, numbers followed by fixity followed by more numbers and their fixities. For example : 0 Fx[=value] Fy[=value] 15 Fy 35 Fy [[dx=value] [dy=value] [dz=value]].
  - **Fx** - If no value is specified the fixity is full in the X direction.
  - **Fy** - If no value is specified the fixity is full in the Y direction.
  - **Mz** - If no value is specified the fixity is full in the Z.
  - **dx** - Sets the support displacement in the X-axis.
  - **dy** - Sets the support displacement in the Y-axis.
  - **dz** - Sets rotation about the Z-axis. Note: that dz should always be in radians.

  If an argument is specified that sets the supports of the design (one beam).
- **release** - A string of the form, numbers followed by fixity followed by more numbers and their fixities. For example : 0 Fx[=number] Fy[=number] 15 Fy 35 Fy.

  If an argument is specified that sets the releases of the design (one beam). **NOTE: For each release specified it breaks the physical member into additional members.**
- **material** - A string which represents the material properties values for the design. The string is of the form, symbols followed by numerical values. The numbers are in terms of the unit type for force and length. For example : E 29000000.
- **section** - A string which represents the section properties of the physical members.

3.2 User Interface

The user interface is constructed on top of the Netbeans infrastructure. The decision was made to save time and gain functionality the would have been otherwise impossible to develop, support, and maintain. In particular, it offered persistence of user data and user preferences.

3.2.1 Input Validation

Because the user interface requires a large number of text fields input validation is critical. There is no way of knowing what the user will enter. The goal of input validation is to only report valid entries from text fields to the shared data structure.

To determine which entries are valid and which entries are bogus, a utility class was constructed which uses regular expressions to determine whether or not an entry is valid.

Most of the string which needed to be validated were lists of nodes written in engineer style notation. For example, "0 Fy Fx 10 Fy" is a relatively simple node list which specifies two nodes. A more complicated example, "0 Fy Fx=12 dx=0.0012 10 Fy Mz=0.00026 25 Fy" specifies three nodes. Although regular expressions worked to specify this language, specifying the rules was not trivial.

3.2.2 Load Combination Evaluator

There are many legal requirements regarding structural analysis. These requirements are constantly changing and differ from region to region. If we were to hard code a solution for each region this would take far too much time. In addition our software would be quickly outdated whenever the requirements were changed for any region.

We need a parser that can read requirements from a file which can be easily modified. In this file we can place many common functions which can be referred to by name. As requirements change, the file may be changed without any need to recompile the software. Thus, our software package is not outdated and new codes can be specified quickly and by the user instead of a programmer.

Although this is a very simple language there is a need for a formal specification.

A grammar is needed to specify which sentences are syntactically correct. Only the syntactically correct sentences can be evaluated.

```
<program>   ::= '(' <id> ')' <statement>
<id>        ::= <alpha> { [ <alpha> <number> ] }*
<statement> ::= <number><word> | '(' <statement> ')' |
    <statement> <operator> <statement>   |
    <function>
<function>  ::= <id> '(' <statement> {', '<statement>}* ')'

<word>      ::= {<alpha>}*
```

Before a sentence is evaluated it must be parsed into more manageable data objects. This section describes the process of parsing a raw string into those objects.

The sentence is first scanned into tokens. The tokens are stored in a tree structure. Each node of the tree is a string or contains references to children. Any place in the string where a matching set of parenthesis are found is treated as a subtree.

For example, the statement "(3.4E + 4.5D) * 3.5Lr" is stored in a list with three elements. The first element is a list with the elements "3.4E", "+", and "4.5D". The second element of the initial list is "*". The third element of the list is "3.5Lr".

Once a statement has been tokenized each "OR" function call is found. A list is returned for each argument of a call to "OR" replaced by one of its arguments.

The evaluation should return the maximum value generated by the specified equation.

3.2.3 Unit Conversion

One of the goals of SAD was to allow the engineer to work in whatever units they wished. This meant that it was up to the internals of the program to keep units straight. With so many conversions going on it is necessary to have a standard way to convert units back and forth from user units to standard units and back.

The unit converter reads a table of conversions from a file and using the convert method allows conversions to and from any units specified by the table. The unit converter also restricts invalid conversions.

3.3 Solver

There are multiple types of structural problems. SAD focuses on one and two dimensional problems, but the general solver, described later, is capable of handling three dimensional problems.

The general solver is capable of solving one, two, and three dimensional structural problems. Once a problem has been parsed into data structures the problem is simple enough that the same solver will work for all problems.

More specialized events are needed to set up the problem in a way that it can be solved by the general solver.

Given a problem description the solver calculates results of various load combinations in accordance with local laws using element stiffness matrices. A local stiffness matrix is calculated for each element. The local matrices then are combined to form a global stiffness matrix. The global stiffness matrix is then used to calculate the displacements of the nodes of members in the design.
The task of the solver can be broken down into the following steps as learned from [1].

1. **Breakdown**
   (a) **Disconnection** - break down the description into elements and nodes.
   (b) **Localization** - convert a elements' and nodes' coordinates to local coordinates.
   (c) **Member Element Formation** - calculate each elements local stiffness matrix.
2. **Assembly & Solution**
   (a) **Globalization** - convert the elements' and nodes' local stiffness matrix to global coordinates.
   (b) **Merge** - merge the local stiffness matrices into one global matrix.
   (c) **Application of Boundary Conditions** - Enter the known forces and displacements into the equation.
   (d) **Solution** - Solve for the unknown displacements or forces.

3.3.1 General Solver

The general solver takes the global matrix, displacement, and force vectors and uses linear algebra to solve the for unknowns.

3.3.2 Beam Solver

The beam solver is really not responsible for solving anything. Its job is to build the internal problem representation that can be solved by the general solver. When the beam solver is notified by the user interface that a complete problem description has been specified, the beam solver gathers the necessary data from the shared data structure.

First the supports are converted into nodes. All of the nodes are sorted and extra nodes are added if needed. The materials and section properties are collected and the nodes are used to construct beam elements.

Each beam element is capable of constructing its own local stiffness matrix. These local stiffness matrices are forged into one global stiffness matrix.

4. CURRENT STATE

The general solver is completely implemented as well as the load combination evaluation language. It is possible to construct a complete beam problem and solve it. The user interface is not yet connected and working with the solver. As of right now all problems must be coded instead of read from text fields in the application.

The results of our unit tests so far have been positive. We have had problems with the representation of double precision floating point decimal numbers. Because of the way these numbers are some of our numbers are off by miniscule amounts. While these numbers are technically wrong, the discrepancy is small enough that it should not matter.

5. CODE DOCUMENTATION

This section describes where things are in the project. Our code is spread across three Netbeans modules, Common, UI, and core. The code in Common are utility classes and other code that should be accessible to both the user interface and core solving routines. The code in UI is related to the user interface. Code in the core module contains parsing classes, the solver, and classes pertaining to internal representations of elements and nodes.

For a more detailed description of our code take a look at the Java documentation generated directly from our source code.

6. 3RD PARTY LIBRARIES

Additional libraries were used to speed development. Jama is used as a linear algebra package. Although it is not optimized for structural analysis it is a fairly complete library written in pure Java. This minimizes the hassles of trying to recompile a 3rd party library for multiple platforms. It is available under the MIT license which makes it free and open more so than the GNU GPL.

Since the project was developed in Java we used a lot of standard desktop libraries such as lists and maps. The Netbeans framework also incorporates a lot of Sun packages.

7. USER MANUAL

Because the user interface portion of our project is not yet connected to the solver a user's manual would not be very useful. At this point we can only describe the basic user interactions.

8. CLOSING REMARKS

This project has turned out to be more experimental than previously anticipated. While we started with some idea of how to approach the problem, the majority of our effort was spent on attempting to exactly describe the break down and solution process in modules simple enough to program.

This project has taken a tremendous amount of cooperation. We have discovered a deep appreciation for the software development process and those who are able to produce quality software.

9. REFERENCES

[1] Robert E. Sennett. *Matrix Analysis of Structures.* Waveland Press, Inc., Long Grove, IL 60047.