An Implementation of a 2D-Truss Analysis Tool in Python

Carlos Carrasquillo*
c.carrasquillo@ufl.edu, UFID: 9579-7665

This paper details the development of a 2D-truss solver using finite element methods. The solver was developed in Python and takes advantage of object-oriented programming to allow for the simple and intuitive addition of element types. The software utilizes a proprietary file format capable of storing the node connectivity table, truss properties, and boundary conditions. The file format is also capable of accepting inputs in both the Cartesian and polar coordinate spaces. Using LU decomposition, the nodal displacements were computed. The resulting nodal displacements are written out to a text file for the user's convenience. The displacements obtained were identical to those obtained in both manual computations and while using established finite element analysis software.

December 9, 2020

*University of Florida Student, Department of Mechanical and Aerospace Engineering, Gainesville, FL, 32611

I. Table of Contents

Contents

II. Introduction

A. Problem Statement

Fig. 1 An image of a roof truss for a house. This image was obtained from 84 Lumber.

Trusses are interconnected structures composed of bar-like elements simply connected at the endpoints. Trusses are often used to support external loads by distributing the forces experienced at the nodes to multiple elements, therefore reducing the stresses endured by each individual element. They are particularly known for having the capacity to resist large loads while requiring low quantities of material. For this reason, trusses are often used in large construction processes, including bridges, stadiums, and buildings among others (Fig. 1).

The paper describes the implementation of a simple 2D-truss solver using the finite element method. The minimum requirements for the implementation are listed below.

- The finite element mesh, boundary conditions, material properties, and other relevant information should be read in from a file.
- The software must able to accept a variable number of truss elements and nodes.
- The resulting nodal displacements should be outputted to a file.
- The software should be modular and scalable such that different element types can be added with minimal changes.

B. The Plane Truss Element

The plane truss, also referred to as the 2D truss, is composed of a set of bar-like elements connected connected only at the endpoints. Usually, the following assumptions are made about truss elements.

- All elements have a uniform cross-sectional area.
- All members are connected to each other or a joint only at their ends by frictionless hinges.
- Each element has a uniform cross-sectional area.
- Loadings and support reactions are only applied at the joints.
- The centroidal axis of each element intersects the nodes at the endpoints.
- No bending moments are applied.

$$\sigma = E\varepsilon \tag{1}$$

$$\frac{d}{dx}\left(EA\frac{du}{dx}\right) \tag{2}$$

The constitutive equation for the truss element is Hooke's Law (1), where σ is the normal stress, E is the material modulus of elasticity, and ϵ is the strain. At equilibrium conditions, the constitutive equation can be expressed as a function of material properties and the displacement of the nodes (2). The weighted residual method can then be applied to this equation to obtain the weak form (3).

$$\int_0^L \frac{d}{dx}\left(EA\frac{du}{dx}\right)\,dx = 0 \tag{3}$$

This result can be integrated by parts to model the truss as a system of linear equations, solvable using elementary linear algebra techniques (4-6).

$$\int_0^L \frac{d\delta u}{dx}EA\frac{du}{dx}\,dx = EA\frac{du}{dx}\delta u|_L - EA\frac{du}{dx}\delta u|_0 \tag{4}$$

$$\int_0^L \frac{d\delta u}{dx}EA\frac{du}{dx}\,dx = F_1\delta u_1 + F_2\delta u_2 \tag{5}$$

Or, in matrix form,

$$\left\{\delta \mathbf{X_e}\right\}^T \left[\mathbf{K_e}\right]\left\{\mathbf{X_e}\right\} = \left\{\delta \mathbf{X_e}\right\}^T \left\{\mathbf{F_e}\right\} \tag{6}$$

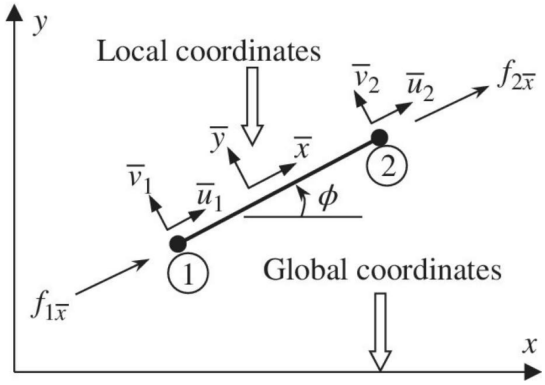

Fig. 2 The local and global truss element coordinate systems [1].

The truss system is defined relative to a ground coordinate system, where the x and y axes are oriented along the width of the page (right) and the height of the page (up), respectively. However, a singular truss element is best defined using a local coordinate system that has axes parallel to and perpendicular to the length. This coordinate system allows the nodal displacements to be provided in the element's native coordinate system.

$$\begin{Bmatrix} \bar{u}_1 \\ \bar{v}_1 \\ \bar{u}_2 \\ \bar{v}_2 \end{Bmatrix} = \begin{bmatrix} \cos\phi & \sin\phi & 0 & 0 \\ -\sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & \cos\phi & \sin\phi \\ 0 & 0 & -\sin\phi & \cos\phi \end{bmatrix} \begin{Bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{Bmatrix} \tag{7}$$

This selection also provides a simple transformation between the local and coordinate systems (7), where ϕ is the angle from the positive global x-axis to the element. Using the direct stiffness method, the stiffness matrix can be defined as,

$$\left[\mathbf{K_e}\right] = \frac{EA}{L}\begin{bmatrix} \cos^2\phi & \cos\phi\sin\phi & -\cos^2\phi & -\cos\phi\sin\phi \\ \cos\phi\sin\phi & \sin^2\phi & -\cos\phi\sin\phi & -\sin^2\phi \\ -\cos^2\phi & -\cos\phi\sin\phi & \cos^2\phi & \cos\phi\sin\phi \\ -\cos\phi\sin\phi & -\sin^2\phi & \cos\phi\sin\phi & \sin^2\phi \end{bmatrix} = \begin{bmatrix} l^2 & lm & -l^2 & -lm \\ lm & m^2 & -lm & -m^2 \\ -l^2 & -lm & l^2 & lm \\ -lm & -m^2 & lm & m^2 \end{bmatrix} \tag{8}$$

where E is the material's modulus of Elasticity, A and L are the length and cross-sectional area of the element, respectively, $l = \cos\phi$ and $m = \sin\phi$.

III. Software Design

The primary design goal of the software was to make the application as scalable as possible. This goal was accomplished by using a five-state finite state machine, with each state responsible for a different core functionality of the software.

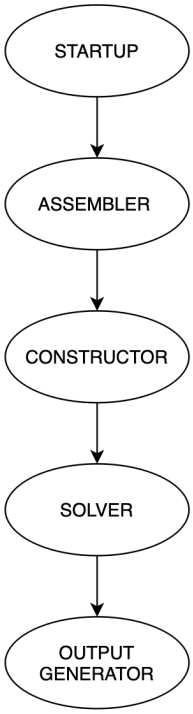

Fig. 3 An overview of the main states in the PyTruss software.

1) **Start Up.** This class is responsible for initializing all of the system parameters, including retrieving the mesh file, the element type, and the file units. Alternatively, this class also allows the user to easily create a new mesh file via an input prompt.

2) **Assembler.** This class is responsible for retrieving the mesh information and generating two iterable lists of element and node objects. The element object contains the connectivity table and the material and geometric properties.

3) **Constructor.** This class constructs the global stiffness and force matrices of the truss. The individual element stiffness matrices are combined using the connectivity table, and the nodal boundary conditions are used to reduce the number of unknowns.

4) **Solver.** This class utilizes a NumPy LU Factorization solver to solve for the displacements of the nodes in the system.

5) **Output Generator.** The output generator is responsible for creating an output file with all of the element type, units, nodal displacements, and comments.

A. Startup

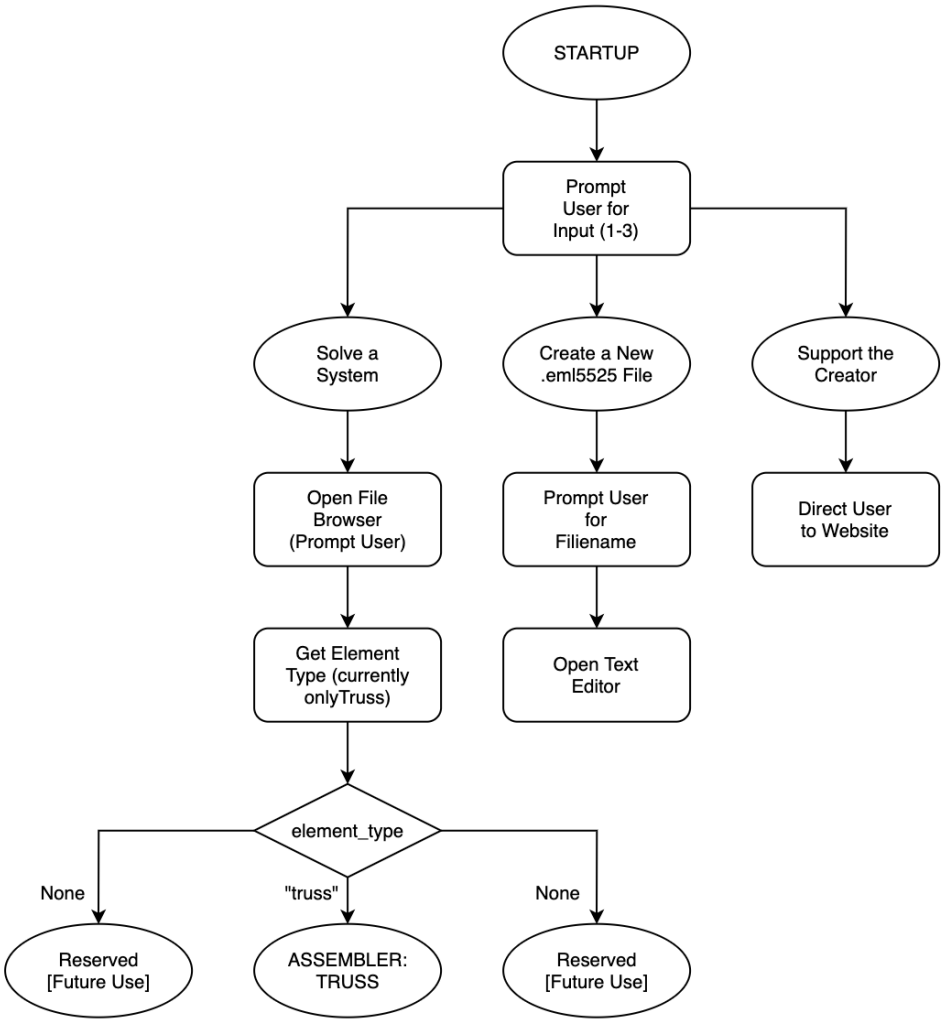

Fig. 4 The flowchart for the software initialization process.

When the program is run, the user is given three options. They can select between solving an existing **.eml5526** file, creating a **.eml5526** file, supporting the creator by visiting their website (the website may take a while to load due to the selected hosting service). The majority of the discussions in this section will relate to the process of solving the truss system provided in the mesh file.

1. Input File Formats

A file extension of **.eml5526** was selected as the input file, but is easily changeable by modifying the **EXT** constant in the *startupProcess.py* file. The file was designed to be as intuitive as possible while still retaining the software's ability to parse the text effectively. The file type was therefore modeled after a simple text file.

```
type: truss
units: IPS
E:1000000, A:5

E1: LN1:1, LN2:2
E2: LN1:1, LN2:3
E3: LN1:1, LN2:4

N1: B:F, FX:1000, FY:1000, XC:0, YC:0
N2: B:FX, XC:-100, YC:173.2
N3: B:FX, XC:-100, YC:0
N4: B:FX, XC:-100, YC:-57.74

Comments:
    - F = Free
    - FX = Fixed
    - VS = Vertical Slider
    - HS = Horizontal Slider
```

Fig. 5 An example of the content of an .eml5526 input file.

The file explicitly declares the element type and the units at the top of the file. No distinction is made between a 2D truss and a 3D truss because future versions of the software will be capable of distinguishing between the two. The third line in the file contains the global parameters. These parameters contain information about the material and geometry of the truss elements. Figure 5 showcases the cross-sectional area and modulus of elasticity constants. If all beam elements are the same length, the length parameter can also be defined as a global parameter.

```
type: truss
units: IPS
E:1000000, A:5

E1: LN1:1, LN2:2, A:6
E2: LN1:1, LN2:3
E3: LN1:1, LN2:4, A:10

N1: B:F, FX:1000, FY:1000, XC:0, YC:0
N2: B:FX, XC:-100, YC:173.2
N3: B:FX, XC:-100, YC:0
N4: B:FX, XC:-100, YC:-57.74

Comments:
    - F = Free
    - FX = Fixed
    - VS = Vertical Slider
    - HS = Horizontal Slider
```

(a)

```
type: truss
units: KMS
A:5, E:1000000

E1: LN1:1, LN2:2, A:6
E2: E:1002000, LN1:1, LN2:3
E3: A:10, LN1:1, LN2:4

N1: B:F, FX:1000, FY:1000
N2: XC:-100, YC:173.2, B:FX
N3: XC:-100, B:FX, YC:0
N4: B:FX, XC:-100, YC:-57.74

Comments:
```

(b)

Fig. 6 (a) Global variables can be overridden by providing element-specific material properties. (b) The order in which the connectivity table, material properties, geometry, and boundary conditions is applied does not affect the readability of the file.

One particularly useful property of the file format is the ability to define element-specific material properties. As shown in Figure 6a, the user is able to specify different areas for each element by appending an extra entry to each element. It is also possible to specify a different modulus of elasticity and length for each element.

The file was designed to be order agnostic. The order of each entry in the lines does not affect how the software processes the inputs. This makes it easy for a user to create a **.eml5526** file without having to frequent the software documentation. The file also has a comments section that is ignored by the assembler and propagated to the output file.

```
type: truss
units: IPS
E:1000000, A:5

E1: LN1:1, LN2:2, L:200, PHI:120
E2: LN1:1, LN2:3, L:100, PHI:180
E3: LN1:1, LN2:4, L:115, PHI:210

N1: B:F, FX:1000, FY:1000
N2: B:FX, XC:-100, YC:173.2
N3: B:FX, XC:-100, YC:0
N4: B:FX, XC:-100, YC:-57.74

Comments:
    - F = Free
    - FX = Fixed
    - VS = Vertical Slider
    - HS = Horizontal Slider
```

(a)

```
type: truss
units: IPS
E:1000000, A:5

E1: LN1:1, LN2:2, L:200, PHI:120
E2: LN1:1, LN2:3, L:100, PHI:180
E3: LN1:1, LN2:4, L:115, PHI:210

N1: B:F, FX:1000, FY:1000
N2: B:FX
N3: B:FX
N4: B:FX

Comments:
    - F = Free
    - FX = Fixed
    - VS = Vertical Slider
    - HS = Horizontal Slider
```

(b)

Fig. 7 (a) Polar coordinate declarations take priority over Cartesian coordinate declarations. (b) Either coordinate system can be input into the file. Only one is necessary for the software to work.

One critical and flexible aspect of the file is the coordinate system declaration The location and orientation of the elements can be defined in one of two ways:

1) **Cartesian Coordinates:** The Cartesian coordinates allow a user to specify the x and y coordinates of each node (Fig 7a). If a Cartesian coordinate is missing and the polar coordinates are not defined, the coordinate will be assumed to be 0.

2) **Polar Coordinates:** The user can also set the length, L, and angle, ϕ, of each element instead of specifying the locations of the nodes (Fig 7b). The element-wise parameters take priority over the nodal parameters, so Polar Coordinates should be omitted if one wishes to use Cartesian Coordinates. As a reminder, the global elemental lengths and angles can be set in the file header.

2. Parsing the File

The file interpreter looks for parameters by searching for the parameter name followed by a colon. For this reason, the user should avoid entering spaces between a parameter name, the colon, and the assigned value. In future iterations of the software, the dependencies on this strict formatting will be reduced.

All parameters, with the exception of the boundary conditions, element type, units, and comments, should be able to be parsed into an integer. If this process fails, the parameter will be set to 0. The supported boundary conditions for a node currently include *free* (F), *fixed* (FX), *vertical slider* (VS), and *horizontal slider* (HS).

B. Assembler

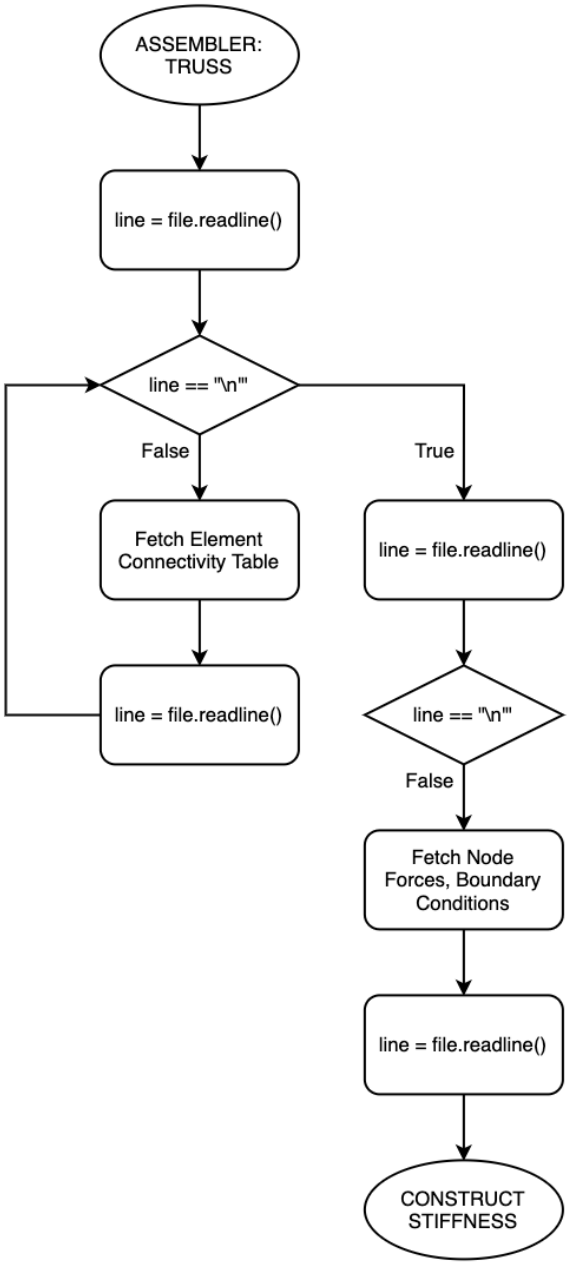

Fig. 8 The flowchart for the software element and node assembly process.

The assembler class is responsible for creating objects for each of the elements and the nodes in the mesh (Fig. 9). The algorithm iterates through the file twice: once for the elements and once for the nodes. The objects are stored in arrays for elements and nodes, respectively, for easy indexing. The element objects contain the following information.

- **LN1:** The first node in the element. If the user is using Cartesian coordinates, the first node is necessarily the point with the smallest y coordinate. If the two elemental nodes have the same y coordinate, LN1 should be the rightmost node.

- **LN2:** The second node in the element. If the user is using Cartesian coordinates, the second node is necessarily the point with the largest y coordinate. If the two elemental nodes have the same y coordinate, LN2 should be the leftmost node.

- **A:** The cross-sectional area of the element.

- **E:** The modulus of elasticity of the element.

- **L:** The length of the element. If it is not specified in the file, it is computed from the nodal coordinates once the nodes are defined.

- **PHI:** The angle of the element, taken from the positive horizontal axis. If it is not specified in the file, it is computed from the nodal coordinates once the nodes are defined.

The node objects contain the following information.

- **n:** The node number/identifier.

- **FX:** The horizontal component of the external loading.

- **FY:** The vertical component of the external loading.

- **B:** The boundary condition. This is a string that instructs the software how to handle the nodal displacements. If unspecified, the boundary condition defaults to *free*.

- **X:** The x-coordinate of the node on the Cartesian plane. This property is not necessary if polar coordinates are selected. If Cartesian coordinates are used and this parameter is not set, the software will default it to zero.

- **Y:** The y-coordinate of the node on the Cartesian plane. This property is not necessary if polar coordinates are selected. If Cartesian coordinates are used and this parameter is not set, the software will default it to zero.

```python
class TrussElement:
    def __init__(self, LN, L, A, E, PHI):
        self.LN1 = check_param(LN[0], integer=True)
        self.LN2 = check_param(LN[1], integer=True)
        self.L = check_param(L)
        self.E = check_param(E)
        self.A = check_param(A)
        self.PHI = check_param(PHI)
```

(a)

```python
class TrussNode:
    def __init__(self, n, FX=0, FY=0, M=0, B='F', X=0, Y=0):
        self.n = check_param(n, integer=True)
        self.FX = check_param(FX)
        self.FY = check_param(FY)
        self.M = check_param(M)
        self.B = B
        self.X = check_param(X)
        self.Y = check_param(Y)
```

(b)

Fig. 9 (a) The element class. (b) The node class.

C. Constructor

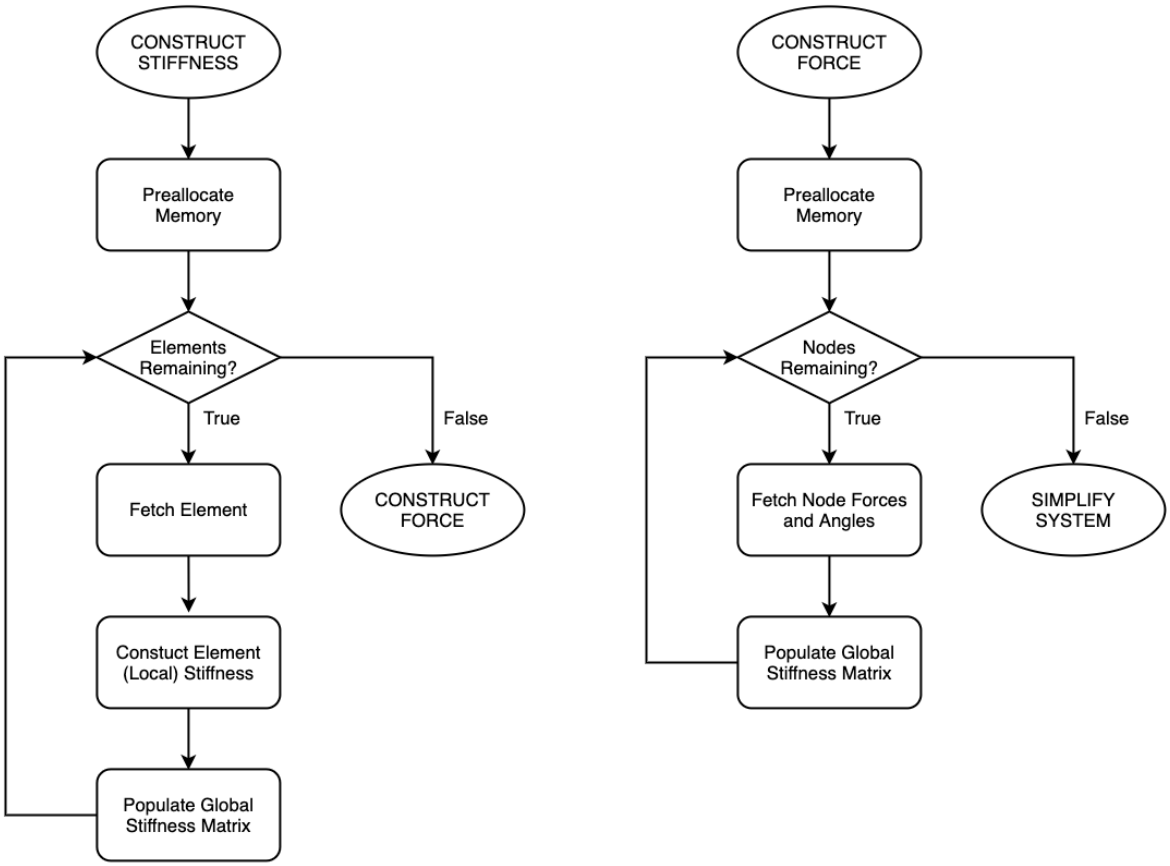

Fig. 10 The flowchart for the global matrix constructor.

The constructor is responsible for creating the global stiffness and force matrices necessary to solve the system of equations (Fig. 10). The global stiffness matrix is constructed by iterating through each element object in the element array and constructing the local stiffness matrices. If the file uses polar coordinates, all of the relevant parameters for the stiffness matrix are obtained from the object properties. If the file utilizes Cartesian coordinates, the coordinates of the element's nodes are used to compute the length of the element and the angle of the element relative to the positive horizontal axis. The element's local node indices in the global truss are then added to the global matrix, which is initialized to zero prior to the iteration.

The global force matrix is constructed similarly, except that the algorithm iterates through the nodes instead of the elements. The algorithm obtains the forces at each node to populate the global force matrix, which is also initialized to zero prior to the iteration.

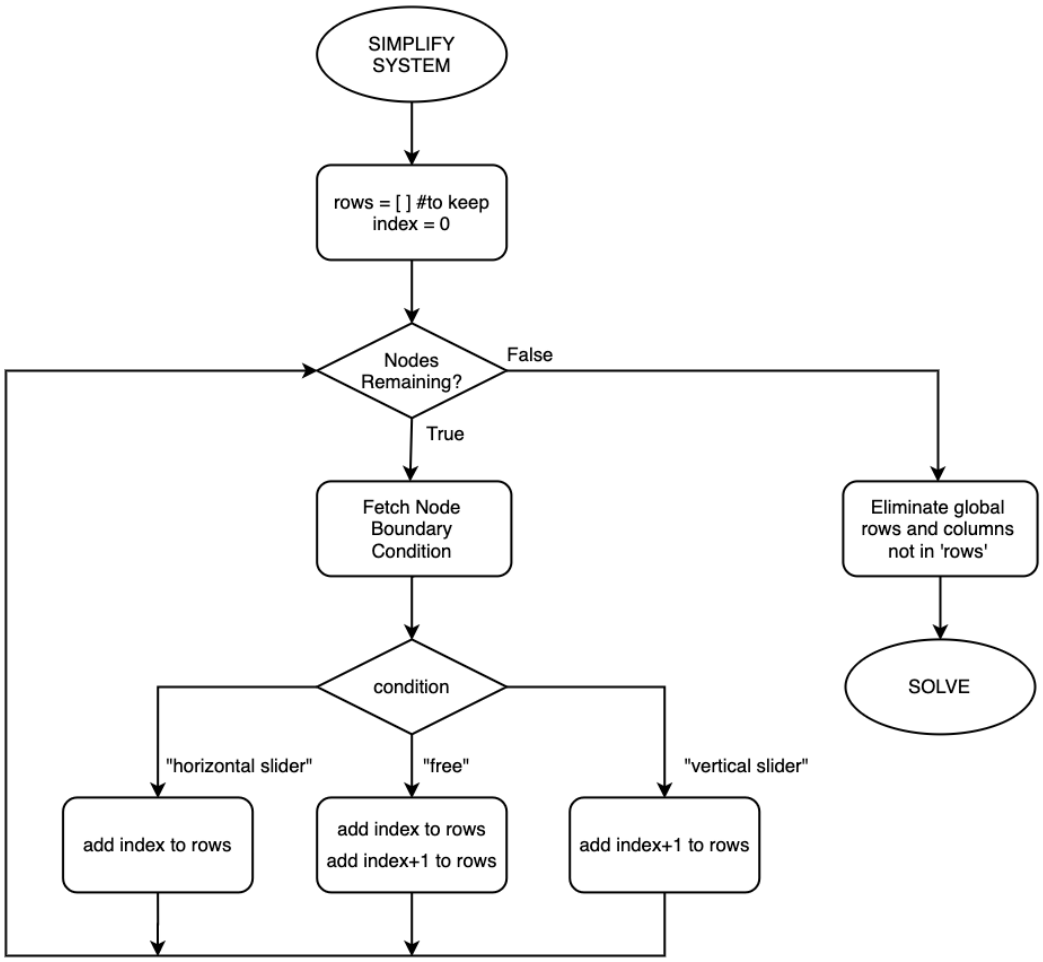

Fig. 11 The flowchart for the global matrix constructor.

The global matrices are then reduced by iterating through each of the nodal boundary conditions (Fig. 11). If the boundary condition has a non-zero component in either of the principal axes, the row index is retrieved and stored in a vector. To reduce the matrix, a counter iterates through the vector of rows to keep and stores them in new matrices. These matrices can then be solved using a systems of equation solver to return the nodal displacements.

D. Solver

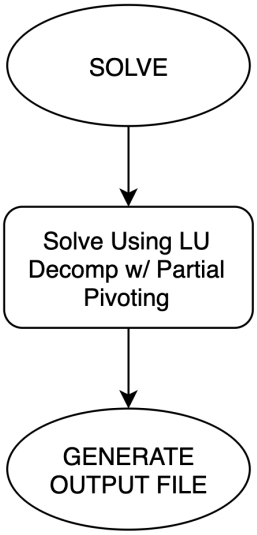

Fig. 12 The flowchart for the solver.

In order to optimize the performance of the software AND improve it's capacity to solve for different loadings, LU factorization was implemented in the solver node. LU factorization is an algorithm that decomposes a matrix into the product of a lower and upper triangular matrix. Since the stiffness matrix only has to be decomposed once, the user can experiment with different loadings on the same truss. The LU factorization algorithm is illustrated as a black box in Fig. 12 because the algorithm is best documented in Dr. Steven C. Chapra's *Applied Numerical Methods with MATLAB for Engineers and Scientists* textbook.

To solve for the unknowns, the LU factorization process decomposes the stiffness matrix into upper and lower triangular matrices. A sequence of forward and backwards substitution processes are then applied to these matrices to obtain the vector of unknowns. This can be accomplished with minimal computational effort because both the forward and back have a computational complexity of approximately $O(n)$ each, or $\sim O(n^2)$ combined (where n is the number of elements in the force vector).

The completed displacement vector is then populated by retrieving the values predetermined by the initial conditions and adding the newly-solved values. The output vector is then passed to the output generator as a parameter.

E. Output Generator

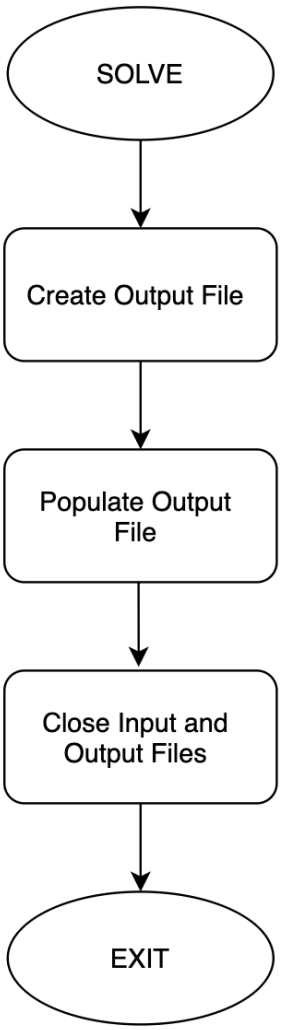

Fig. 13 The flowchart for the output generator.

The output generator is very simple in principle and execution. An output file is created and the filename, units, element type, comments, and displacement vector are written out. Images of the output file are shown in the *Results* section. All files open are subsequently closed and the program exits securely. When this program is developed further, this node will also be responsible for freeing all threads opened to perform the calculation.

IV. Results

To assess the effectiveness of the solver, a number of truss problems with known solutions were tested. One problem was selected to showcase the results. Three variations of this problem are discussed below in further detail.

A. Defining the Problem

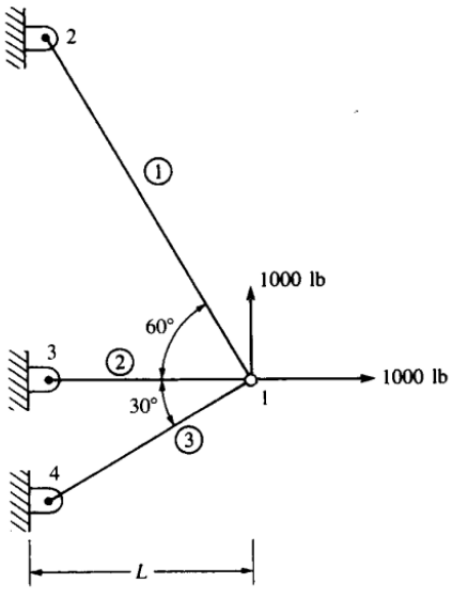

Fig. 14 The 2D truss problem investigated in this section. This image was obtained from EML5526 Homework 4.

The truss system shown in Fig. 14 will be used for reference. Consider the case where $L = 100$, the cross-sectional area of all elements is $5in^2$, and the modulus of elasticity is $E = 10^6$ MPa. The connectivity table for this system is shown in Table 1.

Table 1 Figure 14 Connectivity Table

Element	LN1	LN2	L
E1	1	2	$2L$
E2	1	3	L
E3	1	4	$1.155L$

B. Solving the Problem

Following the guidelines in the *File Format* section, the file can be developed using both the Cartesian and the polar coordinate systems. Both of these examples are shown below.

```
type: truss
units: IPS
E:1000000, A:5

E1: LN1:1, LN2:2, L:200, PHI:120
E2: LN1:1, LN2:3, L:100, PHI:180
E3: LN1:1, LN2:4, L:115, PHI:210

N1: B:F, FX:1000, FY:1000
N2: B:FX
N3: B:FX
N4: B:FX

Comments:
    - Example 1
```

(a)

```
type: truss
units: IPS
E:1000000, A:5

E1: LN1:1, LN2:2
E2: LN1:1, LN2:3
E3: LN1:1, LN2:4

N1: B:F, FX:1000, FY:1000
N2: B:FX, XC:-100, YC:173.2
N3: B:FX, XC:-100, YC:0
N4: B:FX, XC:-100, YC:-57.74

Comments:
    - Example 2
```

(b)

Fig. 15 The mesh file developed for the sample problem using (a) polar coordinates (b) Cartesian coordinates.

It must be noted that, since the problem is most accurately represented using polar coordinates, the Cartesian coordinates were rounded to two decimal places. The program does not have a limit on how many decimal places it can accept (up to 64 bits), but a select few were selected to demonstrate the concept.

```
Element Type: truss
Units: IPS

u1 = 0.008455186619343472
v1 = 0.03155080204459438
u2 = 0.0
v2 = 0.0
u3 = 0.0
v3 = 0.0
u4 = 0.0
v4 = 0.0

Comments:
    - Example 1
```

(a)

```
Element Type: truss
Units: IPS

u1 = 0.008453284038333318
v1 = 0.031545536077074804
u2 = 0.0
v2 = 0.0
u3 = 0.0
v3 = 0.0
u4 = 0.0
v4 = 0.0

Comments:
    - Example 2
```

(b)

Fig. 16 The output file generated for the sample problem using (a) polar coordinates (b) Cartesian coordinates.

The results are accurate to within 0.02%, which can be accounted for by rounding and round-off error. Similar tests were performed on other trusses, with the maximum deviation observed between the choice of coordinates being 0.03% (due mostly to rounding errors). Solving the problem manually and using established finite element software provides the same results. A solution using IBFEM is shown in Figures 17-18. Notice that the maximum displacement occurs at the tip, which is were the corresponding values in the x and y directions are observed.

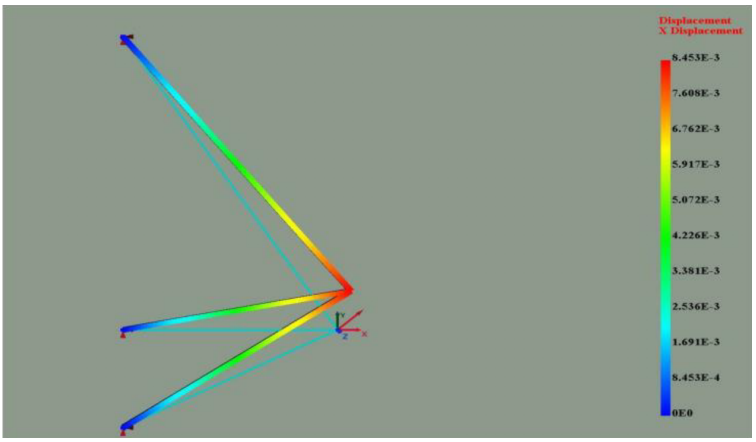

Fig. 17 The IBFEM FEA software solution to the truss structure. The displacement is shown in the *x*-direction. This image was obtained from the EML5526 Homework 4.

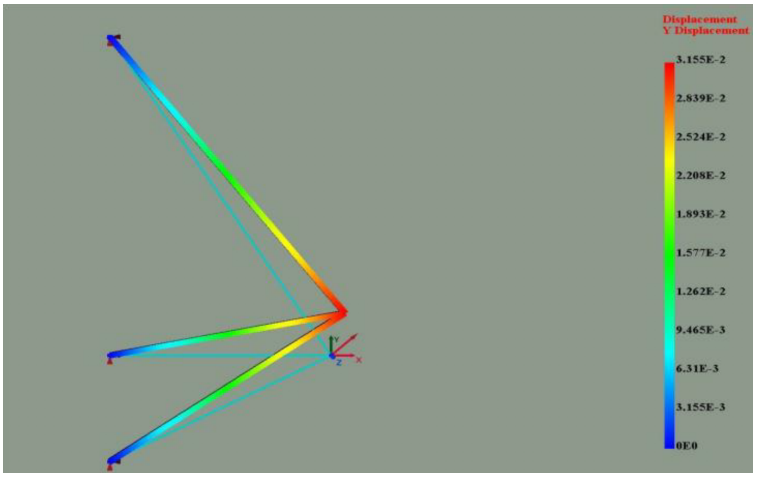

Fig. 18 The IBFEM FEA software solution to the truss structure. The displacement is shown in the *y*-direction. This image was obtained from the EML5526 Homework 4.

V. Discussion

The entirety of the software was developed such that all algorithms, with the exception of the solver, could be computed in $O(n)$ time for an $n \times n$ matrix. The solver adds additional complexity as overhead, which can be shown to be $\sim O(n^3)$ (9)[2].

$$\sum_{k=1}^{n-1} (n-k)\, k\, O(n^3) \tag{9}$$

One of the consequences of this decision is that the program can be very slow with very large mesh files. The benefit of this algorithm is that if the user wishes to test new loading conditions on the same truss, the LU factorization process does not need to be redone. The forwards and backwards substitution processes can run in $\sim O(n^2)$ time.

The solver was also given a standalone node such that, when other numerical algorithms are implemented in the future, the user can easily select the solver. This feature will be particularly useful when iterative matrix solvers are implemented, which can cut down on the computational complexity on much larger matrices.

The dimensionality of the system also affects the computational complexity of the software. While the software currently only supports 1D and 2D truss systems, the software can be upgraded to support 3D trusses by modifying the input file, adding a function call to the interpreter, and modifying the equations used to develop the matrices (this process should take between 5 to 10 minutes). In general, for a truss with n nodes, each dimension adds n new rows and columns to the stiffness matrix. While this does largely contribute to an increase in compute time, the computational complexity of the system does not increase. Although the current implementation does not discriminate between 1D and 2D trusses, future implementations will do so to increase compute times. In order to speed up the program when computing large meshes, users should consider reducing the dimensionality of the problem whenever possible. This may involve considering the displacement in some coordinate axes negligible.

VI. Conclusion

I selected Python as the programming language for this project because I wanted to learn a new language while tackling this project. The experience was incredibly insightful, and I have grown very fond of the language based on the way it handles objects and automatically passes variables by reference.

I was also particularly intrigued by the development of the mesh file. Prior to this project (and this class), I had a disconnect between how the coordinates of the nodes, applied forces, and boundary conditions were used to solve for the unknowns. This was especially the case in 3D models, where the complexity of the parts and selected number of elements can provide very large mesh files. I was also interested in how mesh files were able to adapt to a variety of different element and system types. Building and reading from a mesh file was an invaluable learning experience, as it showed me that, if done properly, it can be very easy to create a file type that is adaptable, easy to interpret (by the computer and user alike), and easy to generate. I plan on refining this software to ensure that it is robust for all likely mistakes in the mesh file, adding new element types, and increasing the supported dimensionality of the software.Eventually, the I also want to build a program with a GUI that will allow a user to automatically generate a mesh file.

References

[1] Kim, Nam H, et al. Introduction to Finite Element Analysis and Design. Wiley, 2017.
[2] Pivoting in LU factorizations. Department of Mathematics, University of Utah, 2017