HIGH SCHOOL EDITION
Journal of Student Research
Volume 12 Issue 1 (2023)

Design and Analysis of Homes for Mars Habitats

Dylan Kang[1], Elsiddig Elmukashfi[#] and Kevin Kukla[#]

[1]Henry M Jackson High School, Mill Creek, WA, USA
[#]Advisor

ABSTRACT

Astronauts need a strong yet portable home design to live on Mars. This paper proposes using an inflatable home with an inflatable carbon fiber internal truss structure covered by a thick silica aerogel tarp to house the astronauts. The internal truss structure is inflatable, i.e., a bar is an inflatable cylinder that contains a central cord, and therefore, it has a variable stiffness in tension and compression where the compression stiffness depends on the internal pressure of the bar. Thus, we introduced alpha ratios as the ratio between the stiffness in compression to tension. To ensure the home's safety, we designed a program in Python that analyzes the home for element forces, reaction forces, and displacements using the direct stiffness method. The design objective is to minimize the deflection of the structure. Three external forces were considered in the program: gravity, internal-to-external pressure difference, and drag force. Taking $\alpha = 0.001$, the maximal displacement was 0.049 m whereas the average displacement was 0.0008 m. These results show that the Mars home design could easily withstand the forces it would experience on Mars.

Introduction

In recent years, going to Mars has become an obsession. In fact, Elon Musk predicts a manned mission to Mars by 2029 (Torchinsky, 2022). Although there have been significant advances in rocket technology, a safe Mars home is still a necessity. Therefore, we created a free and open source program to structurally analyze most two and three dimensional structures that is accessible on Github. We used this program to design an inflatable Mars home and test different materials by changing the element stiffness through the alpha ratio (compression/tension). From this analysis, the ideal inflatable Mars home made of a carbon fiber composite would need an alpha ratio of 0.0005 at the minimum without any element cross-sectional area modifications, although this is only because of a large displacement at a center node of the roof because of low element stiffness in a specific element. Previous works have also created many different designs for potential Mars habitats, including a Mars ice home (Morris et al., 2016). In this paper, the design of a Mars ice home is thoroughly explained, along with a finite element analysis of the structure. This finite element method is again used in another work, this time to fully analyze a three dimensional printed Mars habitat (Park et al., 2020). We use a more specific form of the finite element method, the direct stiffness method, to analyze our Mars home design. Furthermore, to make the design process of a structure more accessible, we add description of the full process behind digitally creating a home, including its optimization. Instead of using preexisting programs to analyze our structure, we create one. The benefits of this methodology are two-fold. Showing how to build a static analysis program and optimize a design lays a total conceptual framework on how to engineer any structure. It allows one to fully grasp the process of structural engineering, for one will understand the underlying physics principles behind design decisions. Also, testing different loading scenarios and element strengths allows more flexibility in the final design, as the most efficient option for element strength can be implemented based on analyzed displacements and internal forces. To detail the process of creating and testing this design, we have structured the paper in accordance with the following outline. Initially, we explain our methods and their

implementation, leading to a presentation of our program's results. These results are then analyzed, and the paper is concluded.

Methodology

In this section, we discuss the procedure we used to design and test the Mars home, starting with the specifications of the Mars home design. After that, we explain how we model the external forces stressing the structure, and we present the direct stiffness method, which converts these external forces into internal nodal displacements and element forces. Finally, we show how the program we made to model any structure uses the direct stiffness method and outputs the results of its analysis.

The Design

General Structure

We designed the structure to have a hexagonal shape since the hexagon offers the most space with opportunities to form triangular supports in the walls. All the supports are pinned, meaning they prevent any translational movement at their node but do not offer resistance to rotational motion. We do not use fixed supports because they would induce bending in the structure, and the pinned supports connect the bottom six nodes of the structure to the ground. With the top half of the structure, we decided upon connecting all of the points to a central location to spread out the forces along multiple bars. While this results in a greater net force taken by the bars since the force on each node must cancel, the average force per bar is much less. Thus, the structure is able to take significantly more force with this hexagonal lattice because the force on a bar causes it to break, not the sum of the forces taken by all elements. The design shown in Figure 1 is the internal bar element structure of the house. This structure will be covered by a silica aerogel tarp. The dimensions of the regular hexagon shaped house are side lengths of 8.65 m and a maximum height of 15 m.

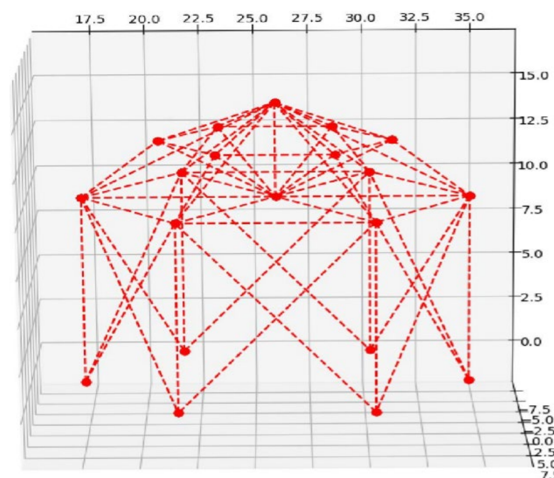


Figure 1. Schematic of the Proposed Design

Maintaining Temperature

The house must keep itself at a reasonable temperature, meaning it should not transfer heat to the Mars atmosphere at an excessive rate through conduction. Fourier's Law of heat conduction governs the process of

this heat transfer which is given by $Q = \frac{kA\Delta T}{t}$, where Q is heat flows through a cross-section of the material per unit time, k is the thermal conductivity, A is the cross-sectional surface area, ΔT is the temperature difference between the ends, and t is the distance between the ends. In this equation, we wanted to minimize Q to keep the inside of the house warm for as long as possible. We did this by minimizing thermal conductivity or k and increasing t or the thickness of the tarp. First, we minimized thermal conductivity by choosing silica aerogel for the tarp, which has a thermal conductivity of $k = 0.013$ $Wm^{-1}K^{-1}$ (Zhang et al., 2018). We also increased the thickness of the silica aerogel tarp to $t = 5$ cm, which is $t = 2$ cm more than necessary (Wordsworth et al., 2019). The density of silica aerogel is $\rho = 0.16$ kgm^{-3}.

The Portability

To send the Mars home to Mars, the home needs to be portable. Thus, we designed it to be inflatable and light, meaning the building's materials needed to be flexible and durable. The natural candidate was a carbon fiber composite, which satisfies both of said requirements. Elements made of this material could be sent to Mars deflated, saving a lot of space during the trip. Then, when needed, the elements would be inflated with a gas, forming the truss backbone of the structure. Astronauts would then attach the silica aerogel tarp to the outside of the truss backbone, completing the structure.

The External Forces

Gravity

We used Newton's Law of Universal Gravitation to find the magnitude of the force of gravity exerted upon the Mars home from Mars which is defined as $F_g = G\frac{m_1 m_2}{r^2}$, where F_g is the gravitational force acting between two objects, m_1 and m_2 are the masses of the objects, r is the distance between the centers of their masses, and G is the gravitational constant. The gravitational forces exerted upon the Mars home by every other object in the universe was considered negligible with a significant r^2 value. For the mass of the Mars home, the mass of the elements was considered negligible, since they are mostly made of gas when inflated. Thus, only the mass of the aerogel tarp is considered. To find this mass, m, we used a rearranged form of volumetric mass density as $m = \rho V$, where ρ is the density and V is the volume. For the Universal Gravitational Constant, the mass of Mars, and the radius of Mars, we used the values 6.6743×10^{-11} $m^3kg^{-1}s^{-2}$, 0.64169×10^{24} kg, and 3.3895 km, respectively (D. Williams, 2022). After determining the magnitude of the force of gravity, we split said magnitude among all the nodes of the structure equally to avoid bending in the elements. The magnitude of the gravitational force per node was 1.5 N, and its direction was straight down.

Pressure Difference

To model the force, F, from the pressure difference between the air inside the home and the Mars atmosphere, we used a rearranged form of the definition of pressure, P, as $F = PA$. For the value of pressure, the pressure difference between the pressure inside the home and the pressure of the Mars atmosphere was inputted. We assumed the pressure inside the home would be 101325 Pa or 1 atm, and the pressure of the Mars atmosphere is around 655 Pa (NASA, 2021). The atmospheric pressure will only be in the bottom part of the house since the roof is sealed off by elements and is inaccessible to humans. Thus, we designed the air in the roof area to have a pressure that is equal the pressure of the Mars atmosphere, meaning there is no force from pressure difference on the roof. This means the lateral surface area added to the area of one of the bases of the hexagonal prism, which is the bottom part of the house, will be the surface area plugged into this equation. We split this force from the pressure difference among all the hexagonal nodes of the structure, and we ensured that each

force would be perpendicular to the node itself and pointing outwards to the Mars atmosphere. The magnitude of pressure's force was 5.0, 521.0, and 362.31 N per node.

Wind

The last force modeled is the drag force, F_d, from the winds on Mars. We used the drag equation to determine the magnitude of the drag force as $F_d = \frac{1}{2}\rho v^2 C_d A$, where v is the wind velocity and C_d is the drag coefficient. When we were entering values in the drag equation, we assumed the worst-case scenario. This was to guarantee that the structure remains safe in any conditions. Thus, we took the highest possible wind speed measured in Martian dust storms as the velocity, which is about 27 ms^{-1} (Mersmann, 2015). The density of the Mars atmosphere at the Mars surface is around 0.02 kgm^{-3} (D. Williams, 2022).

Wind Loading Scenarios

We took three different loading scenarios for the drag force: to the side, to the top, and to both. This was because the wind could be inconsistent. For example, the wind could push solely on the side, or, rarely, it could go over the side of a cliff and push straight down on the structure, and the wind could possibly do both at the same time. When the wind was on the side, we took the drag coefficient of a hexagon with 1 (Aziz et al., 2008). When the wind was on the top, we took the drag coefficient of a sphere with 0.45 because the tarp would balloon from the pressure difference into a sphere (Aziz et al., 2008). These values resulted in a drag force magnitude of 210.42 N per node on the side of the structure and 118.908 N per node on the top.

Direct Stiffness Method

Key Assumptions

In order to use the direct stiffness method (M. Williams & Todd, 2000), we assumed the structure remains elastic, meaning there would be no permanent deformations. We also assumed that all forces would act upon the nodes of the structure, which meant that the elements would not take any bending, allowing us to just use the stiffness of a bar element, k, which is defined by $k = \frac{EA}{L}$, where E is Young's Modulus, A is the bar cross-section area, and L is the bar length. Since the tarp is a continuum structure and thus cannot be analyzed by the truss-based direct stiffness method, we assumed that the tarp always transfers the forces applied to it to the internal truss structure undiminished on the truss' nodes. Finally, we set the cross-sectional area of the bar elements at a constant 1 m^2.

Young's Modulus, E

We built each inflated element as gas inside a carbon fiber composite container. If the element is under tension, the carbon fiber composite container would take the force, so we inputted a Young's Modulus of 250 GPa (Pardini & Manhani, 2003) to the program. However, if the element is under compression, the gas inside would take the force, and there are a variety of possible gases to inflate the element. Thus, we defined a parameter α to find the Young's Modulus under compression as $\alpha = \frac{E_{\text{compression}}}{E_{\text{tension}}}$. We started with α = 0.001 because gases would offer much less resistance than carbon fiber and worked down to 0.0005 and 0.0001 to see how it would affect the displacement of the structure.

Application of the Direct Stiffness Method

Instead of solving for element forces, reactions, and node displacements by hand with the direct stiffness method, we built a program in Python to automate the process (see Figure 2).

Figure 2. The Program's Inputs and Outputs

Key Equation

The key equation for the direct stiffness method is: $f = Ku$. The equation relates the forces, *f*, acting on nodes with the nodal displacements u using the stiffness *K*. There are also two systems of coordinates: global and local. Global coordinates are for the entire structure, while local coordinates are for specific elements. Switching between these coordinates allows us to find the element forces and reactions after calculating nodal displacements.

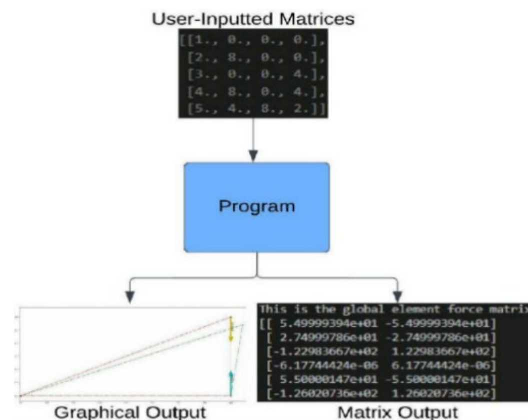
Program Overview

The program is able to, given a two or three dimensional structure, determine the reaction forces, element forces, and displacements of it. It also has certain quality of life features, such as having the option to store data matrices in a JSON file for easy calling and the ability to graph for easier visualization. This program was entirely coded in Python, and the code can be viewed through this link to the public Github repository: Github Code.

Handling Data

The ID System

To keep track of the nodes, each node is assigned an ID. The program assigns these IDs in ascending order. Therefore, node 1 will be the first node inputted, node 2 will be the second node inputted and so on. The identification system is exactly the same for the elements. The count starts at one and goes up until the last element. This is important because the program identifies element node connections through these node IDs instead of node positions. For example, an element matrix might look like this: [1,4,7], where the first number



(1) is the element's ID, the second number (4) is the first node's ID, and the third number (7) is the second node's ID, where the nodes are the ones the element is connected to.

Rounding

The program uses no symbolic math package because symbols are incompatible with numpy's functions. Unfortunately, this causes some inaccuracy in the program because the float64 system, which dictates that numbers use a maximum of 64 bits of memory, needs to truncate numbers. To remedy this, the program rounds a number to the nearest whole one if said numbers are similar within ten decimal places.

Removing Duplicates

Any duplicates in nodes, reaction force nodes, or element nodes would break the program. Thus, the program removes any of the above duplicates it finds in the user-inputted data.

Running Both Two and Three Dimensional Structures

After the program determines if the structure inputted is two or three dimensional, the program creates one of two programmed classes depending on the structure's dimensions. Both classes are specialized for one dimension, since there is no general direct stiffness method that encapsulates all dimensions. Should the structure inputted be two dimensional, the program deletes all data in the inputted matrices relating to the z direction; however, if the structure is three dimensional, the program does nothing with the input data (see Figure 3).

```
1: function CHECKIF2D (nodematrix):
2:     NodeZPos ← EmptyList
3:     for ZPos in nodematrix do
4:         Append ZPos to NodeZPos
5:     end for
6:     if ZPos in NodeZPos all same then
7:         return True
8:     else
9:         return False
10:    end if
11: end function
```

Figure 3. How the Program Handles Two or Three Dimensional Structures

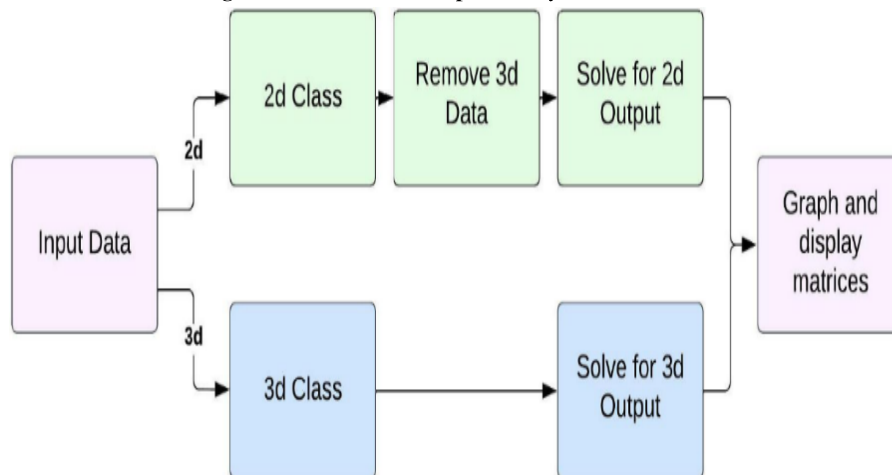
Important General Functions

Choosing a Two or Three Dimensional Representation

The program needs to decide whether the structure inputted is two or three dimensional. In order to determine whether the structure is two or three dimensional, the function extracts the z positions of the nodes from the node data matrix and puts them in a list. Then, the function compares each of the z positions in the list. If they are all equal, the program determines that the structure is two dimensional since the structure would not extend to the z dimension. If they are not all equal, the program returns that the structure is three dimensional (see Figure 4).

Figure 4. Determine if Two or Three Dimensional

Finding an Element's Young's Modulus Multiplied By Cross Sectional Area



To set up an element's local stiffness matrix, the program needs the element's Young's Modulus multiplied by Cross Sectional Area or *EA*. This function asks the user to input a value for *EA*, and it repeats until a valid *EA* is entered by checking if *EA* can be converted to a number. The reason it does this is because if the user accidentally inputs something that cannot be converted to a number, "s" for example, the program would break. The function avoids this by asking the user to input values for *EA* until a valid one is inputted that can be turned into a number. The function then returns said valid value (see Figure 5).

Figure 5. Find EA

```
1: function CONVERT (ConvertMatrix, StiffnessMatrix)
2:     C ← ConvertMatrix
3:     S ← StiffnessMatrix
4:     T ← ConvertMatrix Transposed
5:     Global ← T * S * C
6:     return Rounded Global (3 places)
7: end function
```

Creating the Local Stiffness Matrix

Now that the program has the values for *EA* and *L* (from a function that will be discussed later because it changes based on the structure's dimensions), the program assembles the local stiffness matrix using said values. The function substitutes values into $\frac{EA}{L}$, which is the stiffness of a bar element, and then the function puts these values into the standard local stiffness matrix, which is returned (see Figure 6).

Figure 6. Create Local Stiffness Matrix

Converting Local Stiffness Matrix to a Global One

```
1: function FINDEA (elementname):
2:     repeat
3:         EA ← UserInputEA
4:         if EA can become number then
5:             EA ← EA as number
6:         else if EA can't be number then
7:             Tell User to input valid EA
8:         end if
9:     until EA becomes number
10:    return EA
11: end function
```

To use the stiffness values, the program needs to convert the local stiffness matrix into a global stiffness matrix. To do this, the program performs matrix multiplication using the local stiffness matrix derived in Figure 6 and the conversion matrix, which converts global values to local values for an element. Since the creation of the conversion matrix depends on whether the structure is two or three dimensional, it will be covered later. The function sets up the matrix multiplication necessary by transposing the Conversion Matrix and then goes through with the multiplication. Rounding guarantees that the global stiffness matrix will be symmetric (see Figure 7).

```
1: function STIFFNESS (EA, L)
2:     S ← EA/L
3:     Matrix ← [[S, −S], [−S, S]]
4:     return Matrix
5: end function
```

Figure 7. Convert Local Stiffness Matrix to Global Stiffness

Enforcing Boundary Conditions

To model pinned and roller supports, the program enforces boundary conditions on the assembled global stiffness matrix. Note that the assembled global stiffness matrix is different from the element global stiffness matrix derived in Figure 7. This deletes each corresponding row and column as indicated by the deletion values given in the DeletionList to enforce boundary conditions (see Figure 8). The derivation of said DeletionList depends on the dimensions of the structure.

Figure 8. Enforce Boundary Conditions

```
1: function ELEMENTFORCE (element's displacement, element's local stiffness, ConversionMatrix)
2:     ED ← element's displacement
3:     ELS ← element's local stiffness
4:     CM ← ConversionMatrix
5:     ForceMatrix ← ELS * CM * ED
6:     FM ← ForceMatrix
7:     FFM ← ROUNDING(FM)
8:     return FFM
9: end function
```

Finding Displacements

The program now has all the data necessary to find the nodal displacements with the shaved global stiffness matrix derived in Figure 8 and the external force data matrix. The creation of said force data matrix depends upon whether the structure is two or three dimensional. This function takes the inverse of the Shaved Global Stiffness matrix and multiplies it with the Force Matrix, which returns the Global Displacement Matrix (see Figure 9).

```
1: function SHAVESTIFFNESS (GlobalMatrix, DeletionList)
2:     for Delete Value in DeletionList do
3:         G ← GlobalMatrix
4:         NewList ← Delete Row G
5:         NewList ← Delete Column G
6:     end for
7:     return NewList
8: end function
```

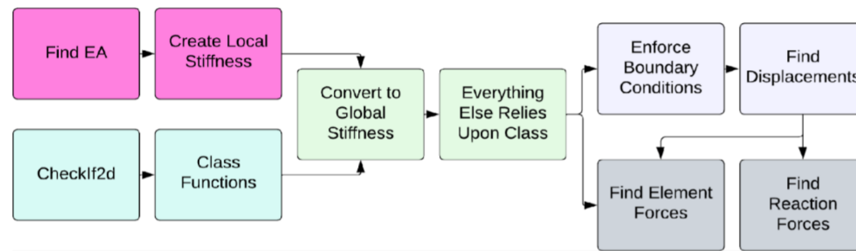
Figure 9. Find Displacement Matrix

Finding Element Forces

Now the program can use global displacements and the stress recovery matrix (local stiffness*conversion

```
1: function DISPLACEMENT (ShavedStiffness, GlobalForce)
2:     Inv ← ShavedStiffness Inverse
3:     FinalMatrix ← Inv * GlobalForce
4:     return FinalMatrix
5: end function
```

matrix), with the local stiffness matrix made in Figure 6 and parts of the global displacement matrix derived by Figure 9, to solve for element forces. The matrix arrangement determines whether the element is in tension (forces stretch the bar) or compression (forces compress the bar). Again, the reason the final force matrix is



rounded is because a symbolic math package is incompatible with numpy, thus the values are rounded to the nearest whole integer should they be sufficiently close (see Figure 10).

Figure 10. Finding Element Forces

Finding Reaction Forces

Now, to find the reaction forces by the supports, the program returns to using the full assembled global stiffness matrix, which is not shaved by Figure 8. This is because the global stiffness matrix was shaved only to determine the displacements at nodes other than where reactions are. Thus, the shaved global stiffness matrix is no longer needed. The program also makes use of the Reaction List, which contains the matrix positions of the reaction forces on the calculated global force matrix and the magnitudes of the external forces acting on the nodes of the reaction forces. Because the matrix positions change with the number of dimensions of the structure, the creation of the Reaction List also depends upon said number of dimensions. For clarification, the same function that creates the Deletion List used in Figure 8 also creates the Reaction List. For each position in the list of Matrix Positions pulled from the Reaction List, the function pulls the total force acting upon a node in a specific direction from the Force Matrix using the matrix position. The function does a similar thing with the external force acting on the node, with a specific external force magnitude being pulled from the Reaction List with the position identifier. To find the reaction force at that node in a specific direction, the function must now subtract the total force acting on the node by the external force acting on the node. This reaction force value and the node it acts upon is appended to a list and then returned (see Figure 11). See Figure 12 for an explanation of the order of use of the general functions mentioned.

Figure 11. Finding Reaction Forces

Figure 12. Order of Usage of the General Functions

```
1: function REACTIONS (GlobalStiffness, ReactionList, GlobalDisplacement)
2:      GS ← GlobalStiffness
3:      RL ← ReactionList
4:      GD ← GlobalDisplacement
5:      ForceMatrix ← GS * GD
6:      GFM ← ForceMatrix
7:      FM ← External Force Mags from RL
8:      P ← Matrix Positions from RL
9:      R ← Empty List
10:     for position in P do:
11:        FR is Final Calculated Reaction
12:        IR ← GFM force identified by P
13:        PFM ← FM force identified by P
14:        FR ← IR - PFM
15:        Append [NodeID, FR] to R
16:     end for
17:     return R
18: end function
```

Dimension Based Functions

Two Dimensions

Because the functions in the two dimensional class are nearly the same as the functions in the three dimensional class (just remove the z direction), we will skip over the two dimensional class.

Getting Element Length and Angle Ratios

To create the Conversion Matrix found in Figures 7 and 10 and the local stiffness matrix found in Figure 6, the program needs both an element's length and angle ratios. The angle ratios are plugged into the Conversion Matrix:

$$\begin{bmatrix} Cx & Cy & Cz & 0 & 0 & 0 \\ 0 & 0 & 0 & Cx & Cy & Cz \end{bmatrix}$$

Instead of finding the angle and then performing trig ratios on it, like $C_x = \cos \alpha$, the program uses the definition of the trig ratio in vector form. For example, the program uses $C_x = x/|r|$. The total magnitude, or the length of the line,
is given by the equation $|r| = \sqrt{x^2 + y^2 + z^2}$.

Creating the Conversion Matrix

Now, with the angle ratios from a function shown in Figure 13, the program substitutes said ratios into the Conversion Matrix.
Figure 13. Find Element Length and Angle Ratios

Assembling the Global Stiffness

The program has all of the element global stiffness matrices from Figure 7. However, the program needs to assemble these matrices into one global stiffness matrix like this:

$$\begin{matrix} & 1x & 1y & 1z & 2x & 2y & 2z \\ 1x \\ 1y \\ 1z \\ 2x \\ 2y \\ 2z \end{matrix} \begin{bmatrix} 11 & 11 & 11 & 12 & 12 & 12 \\ 11 & 11 & 11 & 12 & 12 & 12 \\ 11 & 11 & 11 & 12 & 12 & 12 \\ 21 & 21 & 21 & 22 & 22 & 22 \\ 21 & 21 & 21 & 22 & 22 & 22 \\ 21 & 21 & 21 & 22 & 22 & 22 \end{bmatrix}$$

```
1: function MODELTHELINE (node1, node2)
2:     XM ← x magnitude
3:     YM ← y magnitude
4:     ZM ← z magnitude
5:     VL ← √(XM² + YM² + ZM²)
6:     ARs ← empty list
7:     for mag in (XM, YM, ZM) do:
8:         AR ← mag / VL
9:         FAR ← ROUNDING (AR)
10:        Append FAR to ARs list
11:    end for
12:    return ARs, VL
13: end function
```

Since there are actually four $3x3$ matrices in this single matrix, along the quadrants, the program splits the element global stiffness matrix into said matrices:

$$\begin{bmatrix} \begin{bmatrix} 11 & 11 & 11 \\ 11 & 11 & 11 \\ 11 & 11 & 11 \end{bmatrix} & \begin{bmatrix} 12 & 12 & 12 \\ 12 & 12 & 12 \\ 12 & 12 & 12 \end{bmatrix} \\ \begin{bmatrix} 21 & 21 & 21 \\ 21 & 21 & 21 \\ 21 & 21 & 21 \end{bmatrix} & \begin{bmatrix} 22 & 22 & 22 \\ 22 & 22 & 22 \\ 22 & 22 & 22 \end{bmatrix} \end{bmatrix}$$

$$[NodeID \quad MatrixPos \quad ExternalForce]$$

Since each of the four matrices have constant node data, meaning the nodes do not change, the program can place these matrices into the global stiffness matrix based on the first element in the matrix (top left corner). The initial matrix positions are given by the formula $3 * (NodeID - 1)$ because the matrix positions always increase by 3*s*(1x, 1y, 1z). The function shown in Figure 14 first sets up the four element matrices, splitting the total element stiffness matrix. Then, it makes a list of starting index row and column positions and stores it in the list of *MLs*. Finally, the function iterates over the list of matrices to add them to the global matrix given the matrix positions by *MLs*.

Figure 14. Assemble Global Stiffness

```
1: function ASSEMBLESTIFFNESS (Global, Element, NodeIDs)
2:     E ← Element
3:     M1 ← E[0 to 3, 0 to 3]
4:     M2 ← E[3 to 6, 0 to 3]
5:     M3 ← E[0 to 3, 3 to 6]
6:     M4 ← E[3 to 6, 3 to 6]
7:     F(param) ← 3 * (param - 1)
8:     I ← (F(Node1ID), F(Node2ID))
9:     ML1 ← (I[0], I[0])
10:    ML2 ← (I[0], I[1])
11:    ML3 ← (I[1], I[0])
12:    ML4 ← (I[1], I[1])
13:    MLs ← (ML1, ML2, ML3, ML4)
14:    Count ← 0
15:    for matrix in (M1, M2, M3, M4) do:
16:        Global[MLs[Count]] ← matrix
17:        Count = Count + 1
18:    end for
19:    return Global
20: end function
```

Creating the Global Force Matrix and Deletion List

Now the program must create the Global Force matrix, essential in $f = ku$, and the Deletion List, which is used in enforcing boundary conditions in Figure 8. Fortunately, creating both of these matrices is a matter of expressing the user inputted external force matrix and reaction matrix in a different way. For example, with the external force matrix, all that needs to be done is to go across the row of the matrix and place these in a vertical one, as shown in Figure 15.

Figure 15. Transformation of User-Input to Global Force Matrix

$$\begin{bmatrix} NodeID & F_x & F_y & F_z \\ 1 & 10 & 5 & 0 \\ 2 & 15 & 7 & -3 \end{bmatrix}$$
$$\downarrow$$
$$\begin{matrix} N1 \\ N1 \\ N1 \\ N2 \\ N2 \\ N2 \end{matrix} \begin{bmatrix} 10 \\ 5 \\ 0 \\ 15 \\ 7 \\ -3 \end{bmatrix} \begin{matrix} F_x \\ F_y \\ F_z \\ F_x \\ F_y \\ F_z \end{matrix}$$

Unfortunately, creating the Deletion List is more complex (see Figure 16). The program needs three values per entry in the Deletion List: NodeID, matrix position, and external force magnitude. A NodeID is required to set up the global displacement matrix, matrix position is necessary to delete the correct rows and columns of the global stiffness matrix, and external force magnitude is needed to subtract this from the total force acting on the node to get the reaction force. To determine the matrix position, the program has a matrix position count which increases by 1 every time an entry is added to the global force matrix or the Deletion List.

Figure 16. How the Deletion List Matrix is Set Up

See Figure 17 for the code of the function. First, the function individually accesses and pulls each NodeID from the NodeMatrix to cover the entire stiffness matrix. It then identifies the x, y, and z force magnitudes on the individual node and adds these to individual lists. The reason the function uses lists is because there may be more than one external force entered per node, and this allows the function to efficiently sum the net external forces. The function next pulls where the NodeID equals the Reaction Matrix NodeID (see Figure 18), which allows the function to identify the Degrees of Freedom of the node. If there are no entries in the index list, there are no reaction forces at that node and thus no constraints, and the program just enters the sums of the force lists into the general force list. Otherwise, the program checks each index in the Reaction Matrix to see if the node displacement is constrained (reaction is in the direction) or unconstrained (no reaction in said direction). If the node is constrained, the function adds to the Reaction List; if the node is unconstrained, the function appends to the general force list.

Figure 17. Global Force and Deletion List Creation

Figure 18. An Example User Inputted Reaction Matrix (*Note*. One stands for constrained, two stands for

```
1: function ASSEMBLEFORCES (NodeMatrix, ForceMatrix, ReactionMatrix)
2:     NM ← NodeMatrix
3:     FM ← ForceMatrix
4:     RM ← ReactionMatrix
5:     FL, RL ← Empty List
6:     MCount ← 0
7:     for NodeID in NM do:
8:         TLX, TLY, TLZ ← Empty List
9:         IND ← NodeID=FM NodeIDs
10:        for index in IND do:
11:            TLX append FM[index, 1]
12:            TLY append FM[index, 2]
13:            TLZ append FM[index, 3]
14:        end for
15:        RI ← NodeID=RM NodeIDs
16:        TLT ← (TLX, TLY, TLZ)
17:        if RI length is 0 then
18:            for TL in TLT do:
19:                FL append sum of TL
20:                MCount+ = 1
21:            end for
22:        else
23:            for YI in 1 to 3 do:
24:                if RM[RI[0], YI]=1.0 then
25:                    N ← NodeID
26:                    M ← Mcount
27:                    I ← Sum TLT [YI - 1]
28:                    RL append [N, M, I]
29:                    MCount+ = 1
30:                else
31:                    I ← Sum TLT [YI - 1]
32:                    FL append I
33:                    MCount+ = 1
34:                end if
35:            end for
36:        end if
37:    end for
38:    return FL, RL
39: end function
```

unconstrained.)

NodeID	DOFx	DOFy	DOFz
[1	1	2	2]

Setting Up the Global Displacement Matrix

Figure 9 finds the nodal displacements without reaction nodes. Unfortunately, this shaved version of the Global Displacement Matrix cannot be used to find reaction forces. Thus, the program must fill in a value of 0 for the displacements in constrained nodes to complete the matrix[1]. See Figure 19 for the code of the function. The function first finds out where the reaction forces are on the matrix through the Deletion List, and then it puts 0 on the Global Displacement Matrix (made full of 1s at this point) at said matrix positions. Finally, the function replaces the unfilled parts of the global displacement matrix (parts filled in with 1s) with the displacements given by the Shaved Displacement Matrix.

[1] The displacement is 0 because a directional reaction makes the node remain in place in said direction.

```
1: function SETDISP (NodeMatrix, ShavedDisplacement, ReactionList)
2:     M ← numpy matrix with 1s
3:     NM ← NodeMatrix
4:     SD ← ShavedDisplacement
5:     RL ← ReactionList
6:     for NID in NM[:, 0] do:
7:         IND ← NID = RL[:, 0]
8:         for index in IND do:
9:             M[RL[index, 1], 0] = 0
10:        end for
11:    end for
12:    MC, SC ← 0
13:    for row in M do:
14:        if row[0]≠0 then
15:            M[MC, 0] = SD[SC, 0]
16:            MC+ = 1
17:            SC+ = 1
18:        else
19:            MC+ = 1
20:        end if
21:    end for
22:    return M
23: end function
```

Figure 19. Setting Up Global Displacement Matrix

Extract Element Displacement

To find the element forces derived in Figure 10, the program must extract only the element's displacements from the Global Displacement Matrix made in Figure 19. See Figure 20 for the code of the function. The function starts off by finding the nodes connected to the element, and they are stored in order from left to right, the same way as the setup of the global stiffness matrix. If the program did not keep this consistency, the matrices would multiply the wrong values together and thus result in wrong answers for element forces. Afterwards, the program just adds to the empty list the displacement values of the first and then second node

```
1: function EXTRACTELEMENT (ElementID, ElementMatrix, GlobalDisplacement)
2:     EID ← ElementID
3:     EM ← ElementMatrix
4:     GD ← GlobalDisplacement
5:     M ← Empty List
6:     NIDs ← NodeIDs from EID+EM
7:     for i in 0 to 2 do:
8:         ID ← NIDs[0]-1
9:         M append GD[ID * 3+ i][0]
10:    end for
11:    for i in 0 to 2 do:
12:        ID ← NIDs[1]-1
13:        M append GD[ID * 3+ i][0]
14:    end for
15:    return M as matrix
16: end function
```

with the matrix position identification formula given in Figure 14.

Figure 20. Extract Element Displacement

Graphing

Adding Displacements to Node Positions

To graph the displaced structure, the program must know the node positions after they are displaced. For easier visualization, the program scales the displacements by a factor of two and then adds said displacements to the

```
1. function DISP (NodeMatrix, GlobalDisplacement)
2:     NM ← NodeMatrix
3:     GD ← GlobalDisplacement
4:     DN ← Empty List
5:     DC ← 0
6:     for row in NM do:
7:         T ← Empty List
8:         for v in row[1 to 3] do:
9:             T append v + 2 * GD[DC, 0]
10:            DC+ = 1
11:        end for
12:        DN append T
13:    end for
14:    return DN
15: end function
```

node positions (see Figure 21). This function systematically goes over each nodal position and adds the scaled displacement given by the GlobalDisplacement matrix.

Figure 21. Add Displacements

Plotting the Element Connections and Nodes

Now the program has everything it needs to plot the element connections with nodes represented as bars and dots respectively. The function for graphing the displaced node positions is nearly identical to the function for graphing the nondisplaced node positions; thus, we will only explain the function that graphs the nondisplaced nodes. The program uses matplotlib to graph, and it needs three separate lists in order to graph: xpositions, ypositions, and zpositions. Matplotlib also connects every single node entered in the lists, so the program enters two nodes at a time in lists to be plotted. Otherwise, the program would graph an excess amount of connections. The function's

```
1: function ELEMENTPLOTTING (NodeMatrix, ElementMatrix)
2:     NM ← NodeMatrix
3:     EM ← ElementMatrix
4:     for EL in EM do:
5:         EN1 ← EL[1]
6:         EN2 ← EL[2]
7:         NX, NY, NZ ← Empty List
8:         for EN in (EN1, EN2) do:
9:             IND ← EN=NM[:, 0]
10:            NC = 1
11:            for NL in (NX, NY, NZ) do:
12:                NL append N[IND, NC]
13:                NC+ = 1
14:            end for
15:        end for
16:    end for
17:    Plot NX, NY, NZ: red dotted line
18: end function
```

code is shown in Figure 22. This function appends to each of the individual position lists the positions of the two nodes on the element. The reason a straight line is always graphed is because any curve would introduce a moment, which cannot be accounted for. Thus, we make the assumption that each element is straight.

Figure 22. Plot Element Connections

Setting Plot Limits

In three dimensions, matplotlib does not automatically set the correct plot dimensions, so structures will appear

```
1: function PLOTLIMIT (NodeMatrix)
2:     XL,YL, ZL ← Empty List
3:     for NRow in NodeMatrix do:
4:         XL append NRow[1]
5:         YL append NRow[2]
6:         ZL append NRow[3]
7:     end for
8:     MinL, MaxL ← Empty List
9:     MaxX ← max(XL)
10:    MinX ← min(XL)
11:    Continued for all directions.
12:    for MinV in (MinX, MinY, MinZ) do:
13:        MinL append MinV - 2
14:    end for
15:    for MaxV in (MaxX, MaxY, MaxZ) do:
16:        MaxL append MaxV + 2
17:    end for
18:    return MaxL, MinL
19: end function
```

out of the graph. This is remedied by manually setting the plot dimensions by determining the farthest right and left node positions in all directions and adding or subtracting 2 respectively (1 is manually set as the length of

any vector so that is why the ends of the plot are extended by 2 for safety). See Figure 23 for the code of this function.

Figure 23. Find Plot Limit

Plotting Forces

Now, the program plots the forces acting on the structure. The program shows both the vector itself and the magnitude of the vector. To determine the magnitude of the force vector, the program uses the vector magnitude formula $\sqrt{M_x^2 + M_y^2 + M_z^2}$. Every vector is also scaled down proportionally to a length of 1 to make the vectors easily viewable. The code can be viewed in Figure 24. The first loop in lines 6−9 appends the node positions of

```
1: function PLOTFORCE (NodeMatrix, ForceMatrix)
2:      XL, YL, ZL ← Empty List
3:      XM, YM, ZM ← Empty List
4:      NM ← NodeMatrix
5:      FM ← ForceMatrix
6:      for FN in FM[:, 0] do:
7:          XL append NM[FN-1, 1]
8:          YL append NM[FN-1, 2]
9:          ZL append NM[FN-1, 3]
10:     end for
11:     for FRow in FM do:
12:         ML ← Empty List
13:         for i in 1 to 3 do:
14:             ML append FRow[i]
15:         end for
16:         abs is absolute value
17:         DIV ← Max abs of ML
18:         XM append FRow[1]/DIV
19:         YM append FRow[2]/DIV
20:         ZM append FRow[3]/DIV
21:     end for
22:     PC ← 0
23:     for XPos in XL do:
24:         Y Pos, ZPos ← YL[PC], ZL[PC]
25:         X ← XM[PC]                              ▷ Continue to Z
26:         XMG ← FM[PC, 1]
27:         Continue to Z (1+1...)
28:         PG ← √(XMG² + YMG² + ZMG²)
29:         PC+ = 1
30:         GT ← (XMG, YMG, ZMG)
31:         PT ← (X, Y, Z)
32:         for pos in PT and mag in GT do:
33:             pos+ = mag/2
34:         end for
35:         VecDir ← (XMG, YMG, ZMG)
36:         Plot txt with VecDir as rotation.
37:     end for
38:     Plot vector with XL...+XM...
39: end function
```

every single force to separate lists. Secondly, the loop in lines 11− 21 finds the maximum absolute value of the directional force magnitudes and divides every single force magnitude by said maximum value. This results in all the force magnitudes being scaled down proportionally, with the max dimensional force being 1 or −1. Finally, the function uses matplotlib to graph the vector and a text display, using VecDir as the three dimensional direction to rotate the text. Because the functions to plot external forces and reaction forces are quite similar (same process, just with reaction forces and nodes), we will not cover how the program graphs reaction forces.

Figure 24. Plot Forces

Plotting Element Forces

The program graphs text showing each element's sustained force and whether said element is in tension or compression. It uses the same method as in Figure 24 to determine the necessary rotation of the text to match the

$$\begin{bmatrix} \begin{bmatrix} -4 \end{bmatrix} & \begin{bmatrix} 4 \end{bmatrix} \\ \begin{bmatrix} 10 \end{bmatrix} & \begin{bmatrix} -10 \end{bmatrix} \end{bmatrix}$$

element. This function uses the GlobalElementForces matrix (see Figure 25), which is a list in ascending order (by EID) made from element forces derived by Figure 10. The function (see Figure 26) uses $1P$ and $2P$ as the

```
1: function FORCELABELTEXT (NodeMatrix, ElementMatrix, ElementForces)
2:     NM ← NodeMatrix
3:     EM ← ElementMatrix
4:     EF ← ElementForces
5:     for EID in EM[:, 0] do:
6:         1P ← EM[EID -1, 1] - 1
7:         2P ← EM[EID -1, 2] - 1
8:         NX ← (NM[1P, 1], NM[2P, 1])
9:         Continue till Z: 1+1...
10:        DIVX ← (NX[1]-NX[0])/2
11:        FPX ← NX[0] + DIVX
12:        Continue till Z: NY, NZ
13:        if NX[0]>NX[1] then
14:            RX ← NX[0]-NX[1]
15:            Continue till Z
16:        else
17:            RX ← NX[1]-NX[0]
18:            Continue till Z
19:        end if
20:        LF ← EF[EID -1,0]
21:        RF ← EF[EID -1,1]
22:        if LF>0 then
23:            Plot with Tension
24:        else if LF<0 then
25:            Plot with Compression
26:        else
27:            Plot 0 force member.
28:        end if
29:    end for
30: end function
```

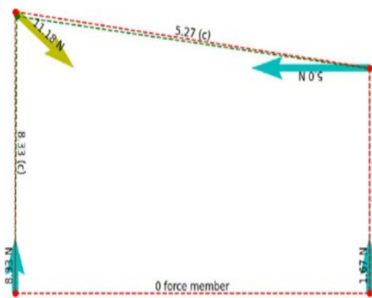
positions of the nodes of the element on the NodeMatrix, which allows it to then pull the nodal positions. Furthermore, the element's length is divided by 2 and added to the initial node position; this is stored as *FPX* and serves as the final position of the text: right in the middle of the element. Lines 13 −18 ensure a positive difference for x, meaning the rotation vector values will be calculated correctly to be entered when plotting. Finally, the function extracts the left side force and right side force and plots based on tension, compression, or 0 force based on these extracted forces.

Figure 25. An Example GlobalElementForces Matrix (*Note.* A negative on the left indicates tension, a negative on the right indicates compression.)

Figure 26. Plot Element Force

Implementation

In this section, we explain the implementation of our program. We start with the testing of the program, and then we discuss how to use it.



Verification

We ran multiple tests in both two and three dimensions. In this section, we will share one test in each of said dimensions.

Two Dimensional Test

This first test was a simple four element structure with three reaction nodes and one external force (see Figure 27). The blue lines were the reaction forces, the red lines were the original structure before displacement, the green lines were the structure after displacement, and the yellow line was the external force vector.

Figure 27. The Program's Two Dimensional Output

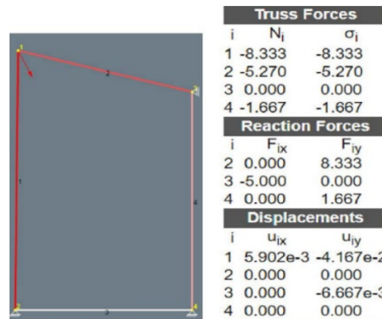


Figure 28 shows the program's text output for the test. We verified the program's results with an online truss calculator (Valdivia, 2019).

Figure 28. The Program's Nodal Displacements Output

See Figure 29 for the online truss calculator's verification. As shown in the figure, the online truss calculator had the exact same calculations for the reaction forces, element forces, and displacements.

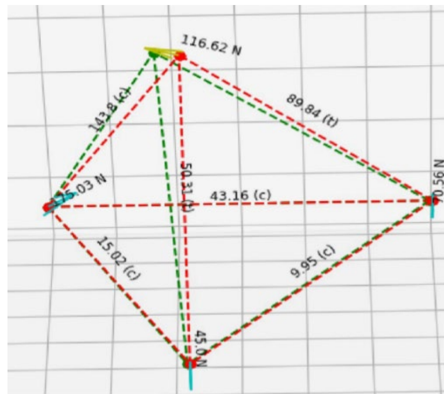


Figure 29. The Online Truss Calculator's Verification

Three Dimensional Test

The three dimensional test was more complex, comprising of four nodes, six elements, and three reaction nodes (Saluague, 2020). See Figure 30 for the program's graphical output. We verified our results with Mr. Saluague's in his Youtube video.

Figure 30. The Program's Three Dimensional Output (*Note.* All Forces are rounded to two decimal places in order to make them easy to visualize.)

See Table 1 for the comparison of reaction forces. Mr. Saluague uses *kN* as his force units, but the program displays *N*. This is not an error because the program just assumes the forces are entered in units of *N*, which they are not in this case. This unit difference does not affect the results of the program. All that happens is that the graphical output says the external force is in *N* instead of *kN*.

```
[[ 0.00590161]
 [-0.04166662]
 [ 0.        ]
 [ 0.        ]
 [ 0.        ]
 [-0.00666671]
 [ 0.        ]
 [ 0.        ]]
```

Table 1. Reaction Forces Compared

	Official Results	My Results
$R1$x	100	100
$R1$y	103.125	103.1249
$R1$z	100	100
$R2$y	−58.125	−58.1251
$R2$z	−40	−40
$R3$y	−45	−44.998

See Table 2 for the comparison of element forces. Unfortunately, Mr. Saluague did not calculate displacements, but since the program uses displacements to find the reaction and element forces, which were both extremely accurate, we can assume that our displacements were accurate also.

Table 2. Element Forces Compared

	Official Results	My Results
$E1$	43.16 C	43.164 C
$E2$	15.019 C	15.0187 C
$E3$	143.801 C	143.80103 C
$E4$	9.95 C	9.949 C
$E5$	89.838 T	89.8383 T
$E6$	50.312 T	50.3113 T

Usage

To use the program, the user must input the node positions, element connections, external forces, and reaction forces of their structure for the program to run. The units of said data are expected to be standard SI. If the user has already stored matrices in the program's JSON file, they may input these into the program. Keep in mind this program does not have a GUI for entering said data, so all data must be entered through text prompt or stored matrices.

Results

$\alpha = 0.001$

Volume 12 Issue 1 (2023)

Given α = 0.001, the maximum displacement over all three wind loading scenarios was 0.0492 m. See Figure 31 for the program's graphical output with an α of 0.001, and see Table 3 for average displacement magnitudes. These values for the max were based off the magnitudes of the displacements. Hence, no negatives were taken into consideration so, for example, −50 *m* would be taken as 50 *m* when determining the maximum displacement. The same principle was applied when we determined the magnitude of the average displacement.
Figure 31. The Program's Graphical Output with the Maximum Possible External Force and α = 0.001 (*Note.* Yellow arrows represent external forces, the green lines represent the displaced structure with a scale factor of 2, the red lines represent the original structure, the red/green dots represent the nodes, the text near an element represents the element's force, and the teal arrows represent reaction forces)

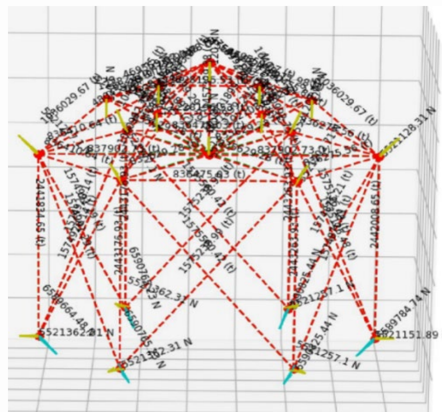
Table 3. The Average Displacement Magnitudes for the Wind Loading Scenarios with α = 0.001

Wind	Avg Displacement (m)
Side	0.000850811
Top	0.000850808
Both	0.000850808

α = 0.0005

Given α = 0.005, the maximum magnitude of displacement was 0.098 m across all three external force scenarios. See Figure 32 for the program's graphical output with an α of 0.0005, and see Table 4 for average displacement magnitudes.
Figure 32. The Program's Graphical Output with the Maximum Possible External Force and α = 0.0005

Table 4. The Average Displacement Magnitudes for the Wind Loading Scenarios with α = 0.0005



Wind	Avg Displacement (m)
Side	0.001669172
Top	0.001669169
Both	0.001669169

α = 0.0001

Given α = 0.0001, the maximum magnitude of displacement was 0.4911 *m* across all three wind loading scenarios. See Figure 33 for the program's graphical output when α is 0.0001, and see Table 5 for the average displacement magnitudes.

Figure 33. The Program's Graphical Output with the Maximum Possible External Force and α = 0.0001

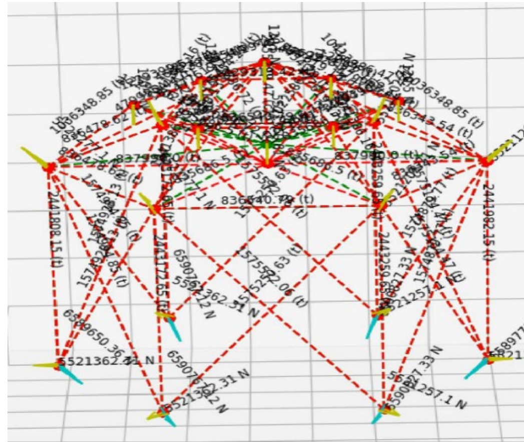


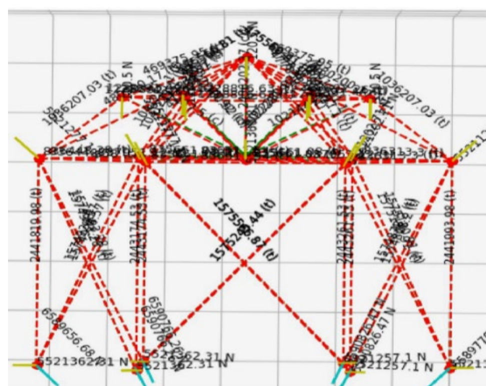
Table 5. The Average Displacement Magnitudes for the Wind Loading Scenarios with α = 0.0001

Wind	Avg Displacement (m)
Side	0.008216038
Top	0.008216034
Both	0.008216035

Element Forces

The maximum element force was 2,455,299.46 N, and this was across all alphas and scenarios. Displayed below (see Table 6) is the average force per element across all scenarios and α values.

Table 6. The Average Forces per Wind Scenarios Across All the Different α Values



Wind	Avg Force (N)
Side	1,172,218.86591
Top	1,172,181.36653
Both	1,172,174.75822

Data Analysis

Given by the average displacement of the structure when under both side and top wind loads with $\alpha = 0.0001$ being 0.008 *m*, the structure will withstand the forces exerted on it by Mars. Although the maximal displacement with $\alpha = 0.0001$ is 0.49 *m*, this displacement only applies to the center roof node, and the displacement was because the element straight above could not easily handle the enormous force of pressure due to its low elastic modulus. This issue could easily be remedied by increasing the cross sectional area of said element to be stiffer in compression and thus allow the node to displace less. The other alphas are clearly capable of withstanding the forces of Mars without modification, with pitiful average displacements of 0.00085 *m* and 0.0016 *m*. Thus, using different alpha scenarios allows more flexibility, as the feasibility of using every alpha definition is known, with only $\alpha = 0.0001$ needing slight modification. This permits engineers to know when and how to change their design when using different gases and materials in the elements without having to perform another static analysis of the structure. Furthermore, the detailed explanation of how to build a static analysis program and start a design from scratch gives a comprehensive conceptual overview of the engineering process. For example, one would understand that to increase the stiffness and durability of a bar, they would need to increase the cross-sectional area or decrease the length. They would also understand that displacements are related to external forces using stiffness, so increasing the stiffness of an element decreases the displacement it endures. Thus, they would be able to effectively optimize their design because they would understand these fundamental concepts.

Conclusion

We used the direct stiffness method to analyze several different loading and alpha scenarios, eventually affirming that most of the alpha scenarios would comfortably withstand Mars' conditions. To do this, we built a static analysis program and then presented exactly how it worked, making it more accessible for others to build their own code and optimize their design. The increased focus on flexibility and accessibility lends itself to discussions about the possibility of simplifying the finite element method. The finite element method is the general form of the direct stiffness method used in this paper, but the finite element method is significantly more complicated. Thus, further research could be done on finding more general and understandable ways of applying the finite element method. Furthermore, the accuracy of the program can be improved by adding compatibility for bending to allow the modeling of gravity, pressure, and wind as uniformly distributed loads.

Acknowledgments

I thank CCIR Cambridge for making possible my learning under my mentor Elsiddig Elmukashfi. I also thank my mentor for helping me with designing the experiment, teaching me everything necessary to perform it, and helping me edit the research paper. Finally, I thank Dr. Kukla for giving me extensive feedback over my research paper.

References

Aziz, E., Chassapis, C., & Esche, S. (2008). *Online wind tunnel laboratory*. https://www.researchgate.net/publication/273771608_Online_Wind_Tunnel_Laboratory

Mersmann, K. (2015). *The fact and fiction of martian dust storms*. https://mars.nasa.gov/news/the-fact-and-fiction-of-martian-dust-storms/

Morris, M., Ciardullo, C., Lents, K., & Montes, J. (2016). *Mars ice house: Using the physics of phase change in 3d printing a habitat with h2o*. https://www.researchgate.net/publication/307965857_Mars_Ice_House_Using_the_Physics_of_Phase_Change_in_3D_Printing_a_Habitat_with_H2O

NASA. (2021). *Mars sequence of events*. https://mars.nasa.gov/MPF/mpf/realtime/mars2.html

Pardini, L., & Manhani, L. (2003). *Influence of the testing gage length on the strength, young's modulus and weibull modulus of carbon fibres and glass fibres.* https://www.scielo.br/j/mr/a/ShzmgmDX8VTSYHHRgSHxbdB/?lang=en#

Park, K., Memari, A., Nazarian, S., & Duarte, J. (2020). *Structural analysis of full-scale and sub-scale structure for digitally designed martian habitat.* https://www.researchgate.net/publication/340393857_Structural_Analysis_of_Full-Scale_and_Sub-Scale_Structure_for_Digitally_Designed_Martian_Habitat

Saluague, A. (2020). *Space trusses*. https://www.youtube.com/watch?v=JoLaoLdsU7g

Torchinsky, R. (2022). *Elon musk hints at a crewed mission to mars in 2029.* https://www.npr.org/2022/03/17/1087167893/elon-musk-mars-2029

Valdivia, A. (2019). *2d-truss analysis*. https://valdivia.staff.jade-hs.de/fachwerk_en.html

Williams, D. (2022). *Mars fact sheet*. https://nssdc.gsfc.nasa.gov/planetary/factsheet/marsfact.html

Williams, M., & Todd, J. (2000). *Structures theory and analysis*. Red Globe Press.

Wordsworth, R., Kerber, L., & Cockell, C. (2019). *Enabling martian habitability with silica aerogel via the solid-state greenhouse effect.* https://www.nature.com/articles/s41550-019-0813-0

Zhang, H., Zhang, C., Ji, W., Wang, X., Li, Y., & Tao, W. (2018). *Experimental characterization of the thermal conductivity and microstructure of opacifier-fiber-aerogel composite.* https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6225116/#:~:text=Aerogels