Tensegrity Space Structures
Static, Kinematic and Stability Behaviour

Muzzammil Sulaiman

Thesis in Partial Fulfillment for the Degree of Master of Science in Engineering

June 2011

Department of Civil Engineering

University of Cape Town

Supervisor: Prof. A. Zingoni

Plagiarism Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own.

Signed by candidate

Muzzammil Sulaiman

Signature removed

Abstract

This dissertation is an in depth study of the design and behaviour of tensegrity space structures. It deals with various aspects of these systems including form-finding and analysis. The dynamic relaxation method is employed to develop a software solution for the design and analysis of tensegrity systems with a focus on its application to irregular forms. This method employs an iterative finite difference approach to integrate nodal parameters over time increments.

Class one tensegrity structures are studied in this dissertation to determine their behaviour characteristics under internal and external actions. Designs are developed, modelled, fabricated and tested under applied loading conditions to validate the results of the numerical model. A four-strut tensegrity module and a three layer tensegrity tower were selected as appropriate for this use. Modules developed are tested independently as well as in assemblies to determine their static and behaviour response under application of external actions. Parameterised models are created to study the effects that certain property variations have on the overall system. Findings were made regarding strut count in diamond tensegrity forms, layer count in tower forms as well as pre-stress effects on load response. It is the aim that the formulations and results presented can be used for further establishment of a pathway for design, fabrication and construction systems for tensegrity structures.

Acknowledgements

This dissertation represents the work carried out for my Masters studies from March 2009 to August 2010 at the University of Cape Town. My studies were conducted under the guidance of my supervisor Professor Alphose Zingoni and funded through the National Research Fund (NRF). I would like to thank:

- Professor Alphose Zingoni for allowing me to complete my Masters studies under his guidance at the University of Cape town.
- Professor Pilate Moyo for his assistance on numerous occasions throughout my studies.
- The Civil Engineering laboratory staff for their efforts in helping me build my models; Noor Hassen, Charles Nicholas and Theo Moyana.
- My friends and research colleagues who made my experiences at UCT enjoyable, also for their valuable help and encouragement; Simon Braun, Bruno Salvoldi, Johnathan Adams, Toni Meri, Patrick Bukenya, Manuel Wieland and Jochen Knecht.
- The NRF for funding my studies.
- Finally, I thank family, especially my parents, for all their support during my years of study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Tensegrity systems are a class of truss structures composed of cables and struts. All connections are pin-jointed and therefore cannot transfer any moments, resulting in axial resistance only. Tensegrity systems differ from ordinary truss structures in that members are either compression resisting; struts, or tension resisting; cables, but cannot be both. The major defining factor for tensegrity systems is that struts are not allowed to connect to each other at any point in the structure. They may only be connected via cables, creating the effect that struts are being suspended in space (Figure 1.1).

Every cable-strut tensegrity system which is conceived is not necessarily in equilibrium, and will collapse into a different structure if constructed. The initial configurations of arbitrary tensegrity structures are therefore unlikely to be stable. Stability is achieved by internal pre-stressing of the elements such that the form is in equilibrium. Determination of this stable self-stress configuration is known as form-finding.

Form-finding is a major challenge associated with the design of these systems. Once the initial geometry has been designed, the equilibrium configuration is to be determined via either model building and experimental pre-stressing of members, or computationally. Each method has its own advantages and drawbacks, and significant work has been done in both areas.

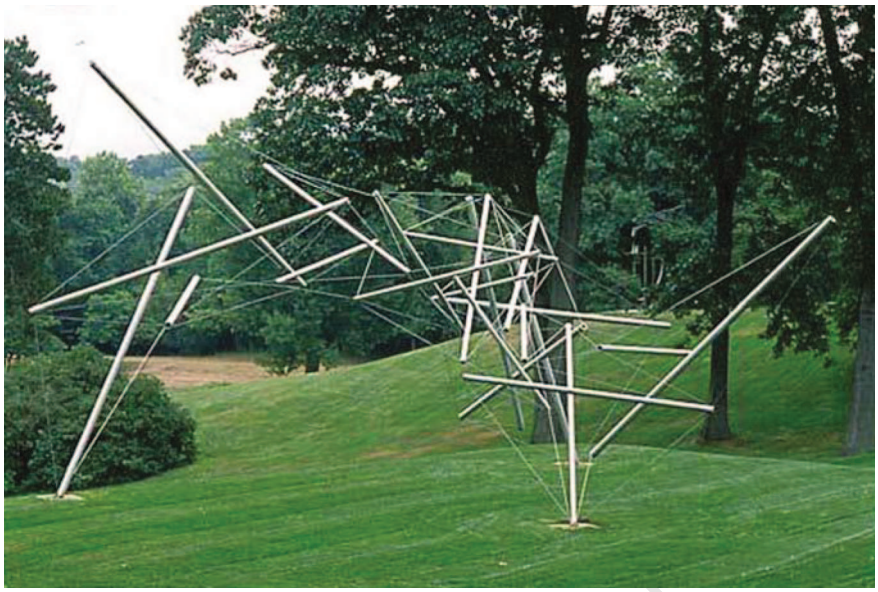


Figure 1.1: Kenneth Snelson's "Free Ride Home", 1974 [1]

Geometric modelling of tensegrity systems is relatively easy, as nodes and struts can be represented mathematically in a number of forms. However, solving the stress states and deformation of the system is the challenging aspect. The reason for this is that the system of equilibrium equations produced is non-linear and consequently computationally expensive to solve. Most methods resolve to linearisation of these equations by introducing restraints, constants or taking advantage of symmetry conditions. These simplifications therefore restrict the tensegrity design to certain forms.

Static and dynamic analysis faces the same problems associated with form-finding. Progress has been made on establishing methods for analysis, including solving the internal forces and deflection by analytic, direct and numerical methods. These methods, however successful, are generally not applicable to large tensegrity systems composed of irregular geometry. This has been an area of increased attention and an effective solution to this problem would provide a path for industrial usage of tensegrity systems and increased practical application.

This dissertation studies the topic of tensegrity structures with a focus on the fundamental concepts required to understand their behaviour. The overall goal of tensegrity research is the design of large, irregular systems, with an eventual pathway to industrial application. The aim of this study is to further the understanding of tensegrity systems and their behaviour, in order to contribute to the overall objective. The geometric design of the topology and configuration is therefore investigated, followed by static analysis including structural rigidity and deformation characteristics.

Chapter 2

Literature Review

2.1 Introduction

A tensegrity system is established when a discontinuous set of compression elements interacts with a continuous set of tension elements forming a stable equilibrium in space [10]. Here continuous implies that members are allowed inter-connectivity, i.e. members of the same type can directly connect to one another. Stability is realised by an internal and external equilibrium. Equilibrium of the elements is ensured by forces in the members. Determination of these internal forces, also known as pre-stress, is the most important stabilising factor in tensegrity systems. Since only axial members are used in these structures, the geometric configuration can be described in terms of nodal coordinates [11]. Determination of the geometry of the equilibrium configuration is most commonly known as form-finding, and in rare instances it is called shape-finding. Form-finding is an integral part of the design process. Most tensegrity systems are underdetermined, thus there is no unique solution to member force magnitudes determined only from the initial configuration [11]. Other geometric restraints, such as symmetry properties, are required in order to produce a unique solution.

Tensegrity structures are lighter and offer a higher mass to strength ra-

tio than conventional compression-based structural systems. Their modular makeup leads to more accurate modelling [12] which can be used to design extremely efficient structures. The node-element design approach leads to a high degree of scalability. Tensegrity structures can be made to be deployable [13], giving rise to further advantages including; transportability, temporary structures, mobility, re-usability. Unlike regular structures, tensegrities can be setup and controlled to allow for large displacements, giving significant advantages for deployable and shape controllable structures [14]. Modular tensegrity systems allow for easy analysis and therefore may result in simpler analytical solutions [15]. This allows large, seemingly complicated structures, to be built up systematically from smaller base modules. The equilibrium position of tensegrity systems may be adjusted by manipulating rest lengths of members and pre-stress forces in members. This allows the same underlying system to have multiple equilibrium configurations, or the position may be adjusted to suit the required application. Essentially this type of construction allows for innovative forms, lightweight structures and deployability [5]. Ali and Smith [16] regard the tensegrity concept, relative to traditional approaches, as one of the most promising for active and deployable structures.

The benefits of tensegrity systems, over classical structural models based on continua of members such as columns, beams plates and shells, is that it allows overall bending of the structure but local bending of individual elements is not allowed [17]. Since only axial forces are transmitted, with no bending or moment resistance in the members, it allows many advantages in modelling. Structural models can be more precise, and better control is allowed via pre-stressing of members. Uni-directional loading implies that no member experiences load reversal, thus reducing certain non-linear problems, such as hysteresis, dead zones and friction, thereby allowing improved control of the system [17]. Tensegrity systems are generally self-supporting and therefore do not need costly anchorages [18].

Very few effective analytical methods exist for computing the geometries

of tensegrity systems [3]. The main challenge in seeking a solution to the form-finding problem is that the resultant system of equations is non-linear. For simple systems, numerical approaches have proven successful, but for systems with more than a few nodes these approaches have been inadequate. Although easy to model, tensegrity systems have proven challenging to analyse due to the large number of variables associated with the system. Nodal coordinates, member lengths, connectivity information and internal and external forces on the system, all have to be taken into account. Authors have developed methods to reduce the number of variables including methods taking into account repetitive forms and symmetry conditions. Others developed models that initially neglected factors such as self-weight and external loading. Although relatively successful on certain geometric forms, the problem of analysing complex geometry with large and irregular arrangement of members still exists.

Challenges in dynamic loading may cause some problems, since tensegrity systems generally have low structural damping [16]. Therefore vibrations may be an issue and should be given more attention in analysis, which means more computational effort.

2.2 Form-Finding

An essential part of the analysis and design of any tensegrity structure is finding the equilibrium geometry. Various solutions to the form-finding problem have been developed, including those based on analytic [19, 8], kinematic [20], static [21, 13], and energy approaches. The first aspect of the form-finding problem is determining the arrangement and connectivity of the system, such that it is capable of structural stability [3]. Once this is found, the geometric and spatial properties of the structure is to be ascertained, including the exact position and rest length of each strut and cable. The form-finding procedures presented previously in literature, for the most part, require certain

assumptions related to member positions and lengths, as well as symmetry conditions, to be known beforehand [5]. The following is a description of the form-finding methods presented in the literature as well as some observations noted by the respective authors.

2.2.1 Analytical

This method uses analytical formulations of node positions, element length and nodal distances to seek a solution to the form of the system. Generally a relationship between lengths of struts and cables, or the angles between them and internal forces, is minimised to find a specific geometric solution. For example, Estrada [8] used tension co-efficients to solve this problem for certain cylinder classes. Bing [22] uses a general analytic method for analysing pin-jointed systems based partly on the Newton iteration method. Prestressability conditions can in some cases be solved analytically as was shown by Sultan et al. [23]. Analytical solutions generally only apply to simple systems with few nodes and members, or with systems with high levels of symmetry.

2.2.2 Force Density Method

This method uses the concept of "force density" to convert the non-linear nodal equations into linear equations [24]. The force density of a member is the ratio of internal member force to member length. Solutions using force density takes it to be a constant for the system, resulting in linearisation of the equilibrium equations. This allows the system to be solved analytically, or by linear or numeric methods. Li et al. [25] reports on research carried out by Motro, who extended the force density with non-linear optimisation to find tensegrity structures subjected to shape constraints. They also discuss Zhang and Ohsaki's approach involving development of an iterative scheme to solve the equilibrium equations using force density. This system can be used for certain structures with irregular topology.

2.2.3 Energy Method

Energy minimisation methods have been used to model and optimise the form-finding of tensegrity systems. A potential energy function is constructed from a given initial topology and system restraints. The total potential energy function should be at a local minimum for a given configuration to be stable [26]. When the endpoints of an element are displaced, energy is transferred to the element and is proportional to the square of the change in length. The energy function is set up such that it is a minimum at the rest length of the element [27]. This allows minimum energy systems to be found, and may be optimum depending on the restraints and initial conditions used for modelling.

2.2.4 Method of Reduced Coordinates

This method regards the struts of a tensegrity system as a set of bilateral restraints acting on the cable system [2]. The position and orientation of the struts are defined, and the the arrangement details of the cable system is calculated relative to this. A virtual work approach is used to calculate the internal forces in the cables. The extension in the struts under the defined arrangement is zero, and therefore their virtual work is also zero. Tibert and Pellegrino [24] outline the major steps for the application of this method. After the generalised coordinates are identified, mathematical software is used to manipulate the system to be able to represent the length of each cable in terms of the defined coordinates. The system is ultimately reduced to a quadratic form and solved either directly for simple systems, or numerically where relevant.

2.2.5 Non-linear programming

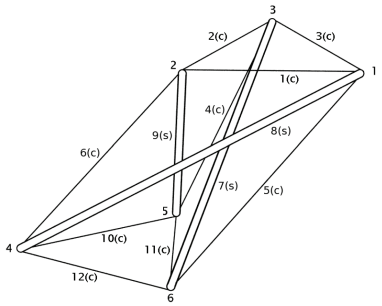
Tibert and Pellegrino [24] describe a general method of form-finding using non-linear programming. Given an initial known topology, i.e. element con-

nectivity and nodal coordinates, this method uses constrained minimisation to determine the pre-stressed configuration. Certain struts are elongated while the length of others are kept constant where required, this is done until the strut lengths are maximised and a stable configuration is found. Burkhardt [28] uses non-linear programming for both design and validation of tensegrity systems, describing a general method for designing skew prisms (See Figure 2.1).

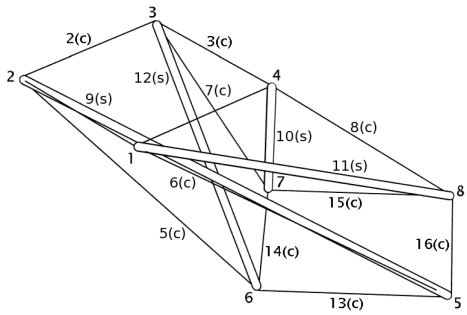
This method does not however take stress constraints into account and even though the resulting configuration may be geometrically consistent, it is not necessarily stable [29]. This approach cannot be applied to larger tensegrities, since as the number of elements increases, the number of constraint equations also increase, which results in a mathematical formulation which cannot be solved directly.

2.2.6 Dynamic Relaxation

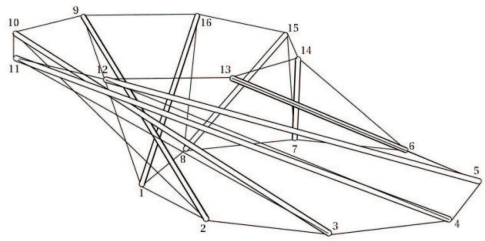
This method was first applied to tensegrity systems by Motro [30] as a general form-finding method. Given the initial topology and external forces, the equilibrium configuration can be calculated by computing a set of dynamic equations. These equations are formulated using the stiffness, mass, damping, external forces, and also the acceleration, velocity and displacement vectors of each of the nodes in the initial configuration. This pseudo-dynamic process can be used to find tensegrity equilibrium positions for irregular systems [25]. The underlying principle in this approach is in modelling the system as a set of masses and dampers. This involves lumping masses at nodes, and calculating damping for each mass-node based on the elements connected to it. It is an iterative solution which allows the system to 'relax' into its equilibrium position, taking into account the application of external loads. Li et al. [25] note Pellegrino's observation that this method may become ineffective for large systems where oscillations occur at certain states in the structure. However, this method has been successfully applied to cable



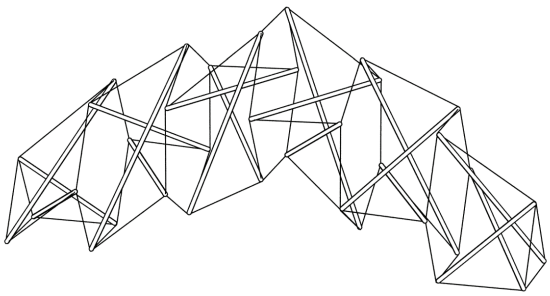
(a) Skew three-strut prism.



(b) Skew four-strut prism.



(c) Skew eight-strut prism.



(d) Skew prism arch.

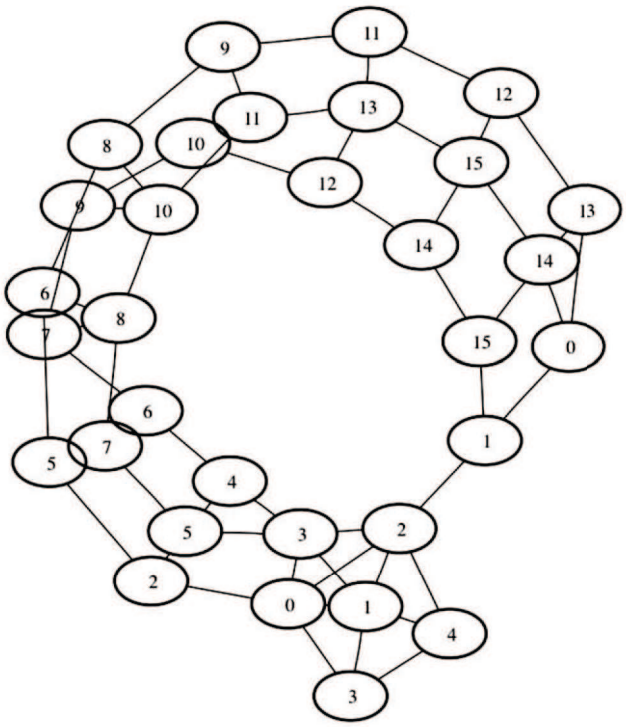
Figure 2.1: Skew prisms designs validated using non-linear programming [2].

systems [31] and its application to tensegrity structures has shown increased appeal.

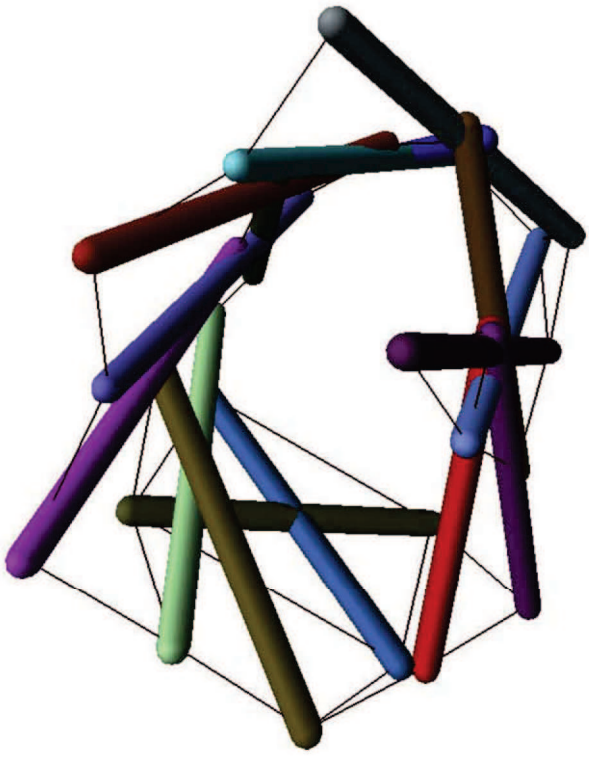
2.2.7 Map L-System

Rieffel et al. [3] use graphs and expand them using a "grammar based developmental process." This system is used to construct large, complex tensegrities from a simple sets of rules governing its expansion. These rules define restraints on attachment of members and nodes to the system, allowing expansion whilst maintaining a resultant system to which the process can be applied.

Map L-Systems operate on edges and nodes of graphs and are used for geometric modelling of the tensegrity structures. A set of rules is defined and a graph is built up using edges and nodes, ensuring that the new system conforms to the rules. A greedy depth-first algorithm can be used to compute candidate pairings of vertices. This algorithm starts with a single node representing an empty set of input attributes. The algorithm decides which attribute should be added to the system as a new node or set of nodes, and its connectivity characteristics [32]. This process is repeated iteratively until no more candidate attributes can be added. The result is a graph (Figure 2.2a) that contains multiple possible alternative pairings, representing different possible tensegrity systems (Figure 2.2b), but not all necessarily optimal. The final system is then left to be computed by an appropriate optimisation approach. The approach adopted by Rieffel et al. [3] is to use an evolutionary algorithm, in this instance, the authors use the Open Dynamic Engine (ODE) . By randomly orienting the structure in the space and setting it free to move from which the engine computes the dynamic equilibrium position. Once the structure is stabilised the simulation is complete. Output variables can then be observed to quantify the level of optimisation and used to compare different optimised configurations. This method has commonalities to the dynamic relaxation method presented earlier in Section 2.2.6.



(a) Graphed solution generated by a map L-system algorithm.



(b) Tensegrity structure corresponding to graph.

Figure 2.2: A 16-strut tensegrity structure generated by map L-system, shown with its corresponding graph [3].

2.2.8 Direct Approach

This approach requires the initial topology and various geometrical restraints to define the form-finding problem such that it can be formulated into a system of linear equations. These equations can then be solved directly to give the member forces and nodal coordinates of the equilibrium configuration. Zhang et al. [11] introduce a process which uses graph theory as tool to model the system and define constraints such as member directions, member types, force vectors and symmetry conditions. If the system is defined strictly enough, so as to allow the equilibrium equations to be written in terms of the components of the member force vectors, then the resulting system of equations will be linear. The incidence matrix is used to manipulate the equilibrium equations in a form representing the member forces. The resulting system of equations can then be solved numerically, and various algorithms are presented by Zhang et al. [11] to demonstrate this.

It has to be noted that member type, i.e. compression or tension, has to be specified initially, which can be problematic since the forces in the system are not known initially. It is therefore the designer's responsibility to define member types, as well as member direction, in terms of the force vector. The procedure can be used generally for checking the existence of the solution to a form-finding problem, although not all solutions obtained may be applicable due to factors such as strut collisions and node congestion. For systems with large numbers of elements and nodes, the system would need to be constrained more in order to achieve linearity of the equilibrium equations. This method may therefore prove impractical for these types of systems.

2.2.9 Monte Carlo

A general Monte Carlo method simulates a stochastic process by applying random configuration changes that are either accepted or rejected according

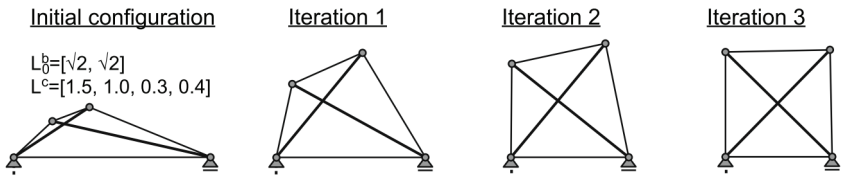
to specified conditions governing a probability distribution [33]. The original approach was developed to use random number generation to compute integrals [34]. It does not require matrix manipulations and is therefore computationally less expensive than most other methods. Li et al. [25] notes that the method, when applied to tensegrity systems, can be used to determine the equilibrium configuration without using any symmetry analysis. Also, that the method may be successfully applied to large scale and irregular systems, without necessarily using geometrical or material constraints. This method uses two assumptions in its application. Firstly, that all the struts and cables are elastic. Secondly, that all the members are connected at nodes via frictionless ball joints. Additionally, self-weight of the members and any damping effects are neglected on the final equilibrium position.

Li et al. [25] also showed that the procedure can find tensegrity forms without the initial input of nodal positions. The method is advantageous as it only uses simple algebraic operation and thus allows easier computational modelling and implementation.

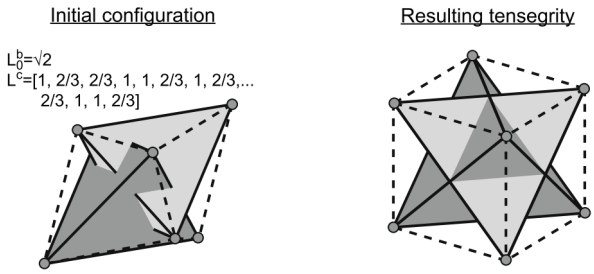
2.2.10 Finite Element Method

A form-finding method that uses the finite element method was presented by Pagitz and Tur [4] to calculate the equilibrium geometry of a structure by iteratively adjusting the lengths of cables. The method requires the topology of the structure, the undeformed strut lengths, total cable length, pre-stress of the cables and stiffness of the struts. The total length of cables is fixed with any adjustments in length of one cable being compensated for by the rest of the cables. The authors present both two and three-dimensional examples to demonstrate the usage of the method (Figure 2.3).

The restriction on this method, in terms of total cable length, result in its usage being limited to structures whose geometry can be predicted beforehand. Predefined cable tension is rarely known before the form-finding process and its requirement here is a disadvantage to the applicability of this



(a) The first three iterations of a two-dimensional two-strut tensegrity structure.



(b) Initial and equilibrium geometry of a three-dimensional tensegrity structure.

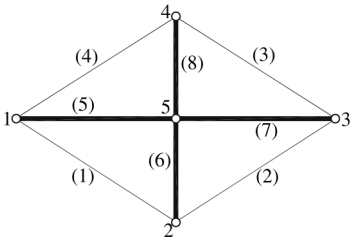
Figure 2.3: Examples of form-finding using the Finite Element Method [4].

method as a general form-finding tool. However, in deployable tensegrity structures, fixed cable length may prove advantageous, by allowing continuous cables to be used and extended or contracted to collapse and deploy the system.

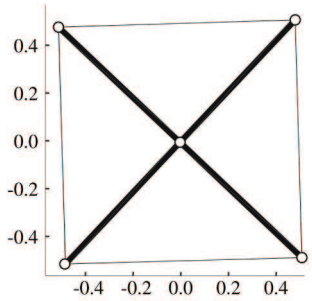
2.2.11 Numerical Method

This numerical form-finding procedure does not require initial nodal coordinates, member lengths or any symmetry conditions to be known. The equilibrium condition for the tensegrity system is computed based on initial topology only [35]. Only the type of member needs to be defined in the topology for this method to be utilised. The roots of this method can be traced back to the force density method.

The incidence matrix is used to derive the force density matrix. Initial



(a) Initial geometry.

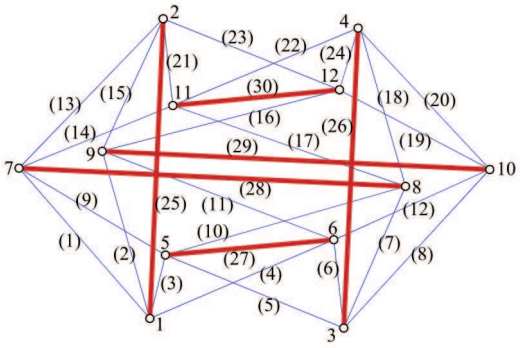


(b) Calculated form.

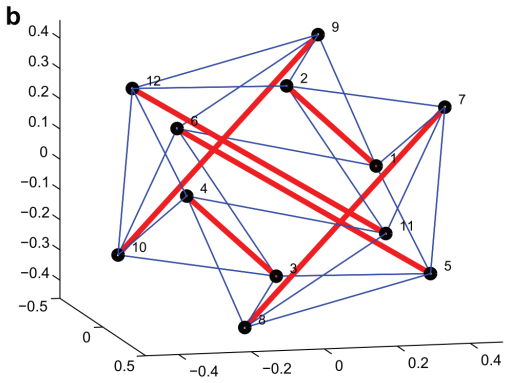
Figure 2.4: Numerical form-finding solution of a two-dimensional tensegrity structure [5].

force densities are assigned based on prototypes, i.e. predefined formulations for computing force-density coefficients based on initial geometric form. This is an iterative process which converges for basic tensegrity forms. The iteration involves decomposition of the force density matrix and equilibrium matrix in order to find admissible nodal coordinates and force densities, which satisfy the minimum requirements for successful computation of solutions.

Tran and Lee [5] only apply this approach to basic tensegrity modules to which they present numerical solutions (See Figure 2.5). However, it is not clearly shown whether this method has potential application to irregular or larger tensegrity systems or how this problem may be approached.



(a) Initial geometry.



(b) Calculated form.

Figure 2.5: Numerical form-finding solution of a three-dimensional expandable octahedron tensegrity structure [5].

2.3 Analysis

In analysing tensegrity structures, many different approaches have been undertaken, including static analysis [29, 36, 21, 13], dynamic analysis [21, 13], kinematic analysis [13, 12], mobility analysis [12], and combinations of static and kinematic analyses [13]. The following is a description of the major methods of analysis applied to tensegrity systems.

2.3.1 Static Analysis

This investigates the behaviour of the system or structure under static loading. Various methods have been attempted in providing a solution to this type of analysis for tensegrity structures, the primary ones being the virtual work method [36], analytical solutions, static analysis of mechanisms, and direct and inverse static problems [12].

Kebiche et al. [37] describe the process for complete analysis of tensegrity systems in three steps. The first step is form-finding, excluding self-stress and external loading, and the subsequent steps are then self-stress determination and then analyses for the application of external loads. The method used to study the behaviour of tensegrity systems is based on geometric non-linear analysis. Large deformations may occur under external actions, resulting in non-linearities in the deformation behaviour. This deformation may be a result of material properties or geometric properties. These geometric properties resulting from the procedure are therefore also taken into account.

2.3.2 Dynamic Analysis

Modal analysis using linearised equations of motion [21], as well as planar dynamic analysis [13] of tensegrity systems have been investigated. Sultan et al. [38] state that research into tensegrity structures should focus more on the dynamics and control systems. They maintain that in the future, tensegrity principles will be applied mostly to controllable structures that

require large motions, thus exploiting the apparent advantages in the areas of flexibility and deployability. A thorough investigation into the dynamics and control of tensegrity structures in therefore essential. However, there have been few studies dedicated to studying the dynamic behaviour of these systems [16].

Tensegrity systems, due to their lightweight nature, are sensitive to dynamic loading and vibration response. Ali and Smith [16] note that studies are usually either analytical or numerical, and few experimental studies have been carried out on full scale models. The result of this is the lack of practical application potential for industrial purposes.

Bossens et al. [39] investigate dynamic effects and responses of a three module tensegrity system, built from basic three-strut prism units. They conduct experiments on the system and use the data to compare the results with that of a finite element model. Experimental data is obtained by placing the structure on a shaker table, and exposing it to different vibrations. Results are compiled from data output retrieved from multiple accelerometers placed on the structure. The results are compared to a finite element model to validate the modal frequencies measured versus those calculated. The authors conclude that a linear model can be used to compute certain dynamic properties of a tensegrity system.

The vibration control and dynamic behaviour of a full-scale active tensegrity structure is studied by Ali and Smith [16]. They, like Bossens et al. [39], identify the resonance modes of the structure experimentally and compare it with results obtained from a finite element model. Dynamic excitation is used to test the structure and identify the dynamic behaviors. Tests were carried out for different self-stress levels at multiple excitation frequencies. Adjustment of the lengths of certain active struts allows for vibration control. By modifying the stress level in the structure, the natural frequencies are moved away from excitation. The experimental and numerical results proved this for different stress states of the structure.

Skelton et al. [17] presents an analytical formulation to model the non-linear dynamic behavior of tensegrity systems. Non-linear equations of motion are derived for deployable tensegrity space structures. The positions and velocities of the ends of struts are used to describe the kinematics of the system, thereby avoiding the use of angular velocities. An analytical expression for the accelerations of all struts is formulated to allow the model to be efficiently simulated, since inversion of the non-linear mass matrix is not required.

2.4 Control of Tensegrity Systems

Active control systems can be utilised by tensegrities, which may allow them to adapt to their environments [7]. A framework of intelligent-control methodologies was used by Adam and Smith [7] to investigate an adaptive structure. The control methodologies utilised were self-diagnosis, multi-objective shape control, and reinforcement learning, which adapted the framework in a tensegrity structure to support its control in certain environments. For practical control purposes, loads cannot be completely defined and therefore should be partially defined in terms of type only, and not magnitude nor location. Self-diagnosis is then required to determine load magnitudes and locations. After the required computations are performed, control commands are then applied to the structure. Reinforcement learning allows the system to adapt to interactions making use of previous control actions stored in memory. This improves performance of the system and allows it to react to loading and other interactions in reduced time. This basic control framework can be adapted to more complex loading, but requires improvements in certain areas in order to be successful in that regard. These areas include first the accommodation of different loading conditions, as well as search and replacement strategies for the control systems.

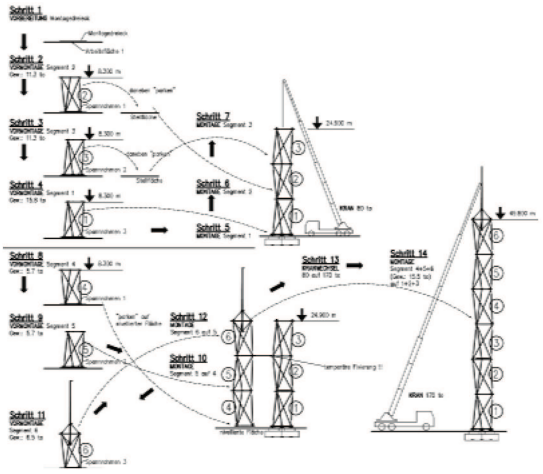


Figure 2.6: Erection plan for tensegrity tower built in Rostack Germany [6].

2.5 Fabrication and Construction

Klimke and Stephan [6] report on the construction of a tensegrity tower in Rostack Germany (Figure 2.6). Challenges faced by the fabrication and construction teams were significant. Among these were large deflections of members due to pre-stressing of cables. This is a typical feature of tensegrity systems and consideration of this needs to be taken account in design. For the required geometry to be achieved, all aspects affecting the final member positions and member forces, need to be taken into account. High tolerances specified by the design, in order to achieve the final member pre-tension forces, were not possible to accomplish. A process of tolerance minimisation was then used to reduce the error as much as possible. Shrinkage due to welding, temperature changes, cable connector shrinkage and cable creep were also noted as additional factors which required design considerations.

A tensegrity structure was built by Adam and Smith [7] and used for experimental testing (Figure 2.7). The structure was equipped with sensors and actuators to adjust or hold particular configuration under the application

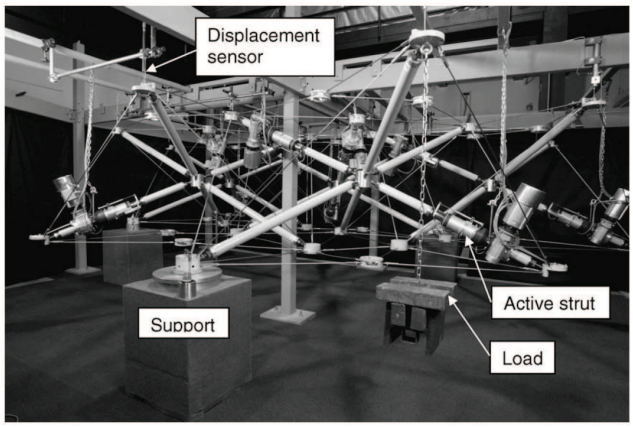


Figure 2.7: Five module, ten-strut actuated tensegrity structure [7].

of load. Multiple active struts allowed strut length adjustment and shape control.

A prototype adjustable tensegrity consisting of three six-strut modules was built by Fest et al. [40]. The structure featured a central joint in the compression members to reduce the buckling length. With the use of telescopic struts and specially developed actuators, the configuration of the system could be adjusted to subsequently change member forces. To allow struts higher degrees of movement, the authors used a central joint in each module. Struts were connected to this joint via ball bearings and allowed rotation in all three axes.

Vyas [41] constructed a poultry shed using simplex units built from bamboo struts and rope. The connections at the ends of struts comprised of steel hooks joined to the strut via epoxy adhesive. Rope tied to these hooks, and interlocking hooks formed the two types of joints utilised. Simplex modules were constructed using these joints, and assembled to form columns and a roof, which were ultimately assembled into the completed structure.

A bi-directional industrial grid was created for prototyping at Nimes, France in December 2000 [9]. The purpose of the project was to assess the feasibility of simple tensegrity systems. Motro [9] notes other tensegrity

structures that are being studied; Kono's double layer grid, Passera and Pedretti's work for the 2002 Swiss expo (including a tensegrity node design), and Paulo Podio Guidugl's arch prototype.

Additional factors to be considered when considering a design for construction are; pre-stressing application, support conditions and how they affect internal force distribution, strut congestion and connection eccentricities, as well as structural stability during assembly and construction.

Construction of tensegrity structures is not as common as conventional structures, but the use of it is increasing. At present, tensegrity is a viable design option when innovation and creativity is a necessity. This necessity can be a result of design constraints, such as was the case for the Kurilpa Bridge in Brisbane, Australia [42]. This pedestrian and cycle bridge had strict design requirements and geometric challenges, including provision of river navigation clearance and relatively flat grades on the site. The success of this project showcases the advantages of tensegrity systems as well as the fabrication and construction possibilities.

There are many challenges associated with tensegrity system implementation. However, the advances in lightweight material technology, improved accuracy in fabrication techniques and the increased acceptance of new design systems in construction industries, provide opportunities and possible solutions to the fabrication and construction problems of tensegrity structures.

2.6 Summary

Tensegrity systems have been extensively studied by many researchers, and the literature on the topic is vast. Many different aspects have been investigated, various problems identified and attempts made to solve them. Analysis of tensegrity systems is one of the fundamental problems that the literature investigates. Authors have applied different techniques and imple-

mented various algorithms in the search for a practical solution. One common theme of these attempts have been to use the geometry to find relationships between members and to use these to solve the equilibrium forms and forces. These techniques try to simplify complex tensegrity geometries such that the system can be mathematically solved.

The advantages of using tensegrity for various applications have been explained by many researchers, but investigation into its practical implementation has not been given in-depth attention in the literature. However, this structural system has been used in construction for multiple projects, and a few of these have been mentioned in this review. Practical construction methods have not been extensively researched and this is an area that requires some further investigation.

In general it can be seen that analytical solutions to tensegrity systems are only available for relatively basic forms. More complicated tensegrities have been investigated using methods that restrict some of the properties or constrain the geometry of these systems. There is thus a need for a robust method for analysing generic tensegrity systems. The dynamic relaxation method shows the most promise in this regard. It has been reported to produce satisfactory results for certain forms and further investigation of its usage on more complex geometry could prove highly useful.

2.7 Aims and Objectives

The main challenges for tensegrity systems are two-fold; structural stability and structural build-ability. Firstly, the geometry needs to be defined such that it is structurally stable in its equilibrium form as well as under applied loading. Once the system is defined, analysed and found to be stable, it is to be designed and detailed so that construction is both possible and practical. Both these problems find their place in the engineering design process and addressing these issues is key to progress in the field of tensegrity structures.

In this process the first step would be to find or develop a suitable design method for these structures. The aim of this dissertation is therefore to investigate this problem and to find possible solutions. Once this is achieved and the method found to be adequate, the second step therefore follows; to determine practical fabrication and construction methods to allow tensegrity structures to be built and function as intended. This second step, although touched upon in this dissertation through the construction of models, is not the main aim of this research. The methodology employed is primarily focused on the design and analysis of tensegrity systems, with the intention of understanding their behaviour under equilibrium and load-bearing conditions. The objective being to add to the research being done in this field, thereby moving towards an overall solution to the design and construction of this relatively new type of structure.

Chapter 3

Basic Concepts

This chapter introduces the basic theory of tensegrity systems. It explains the fundamental concepts and basic principles. All relevant background information and foundational aspects are summarised to introduce the topic of tensegrity and to place the focus area of this dissertation into context. Unless otherwise stated, the material presented below has been sourced from Skelton and de Oliveira [1], Burkhardt [43], Bing [22], Motro [9], Skelton et al. [44].

3.1 Introduction

Tensegrity is defined by Pugh [10] as:

"A tensegrity system is established when a set of discontinuous compressive components interacts with a set of continuous tensile components to define a stable volume in space."

Tensegrity structures are characterised by the way forces are distributed in them. Members are either in tension or in compression. In this thesis, the tension members are cables and the compression members (struts) are circular sections of tubing. The tension members are interconnected forming a continuous network, thereby transmitting tensile forces throughout the

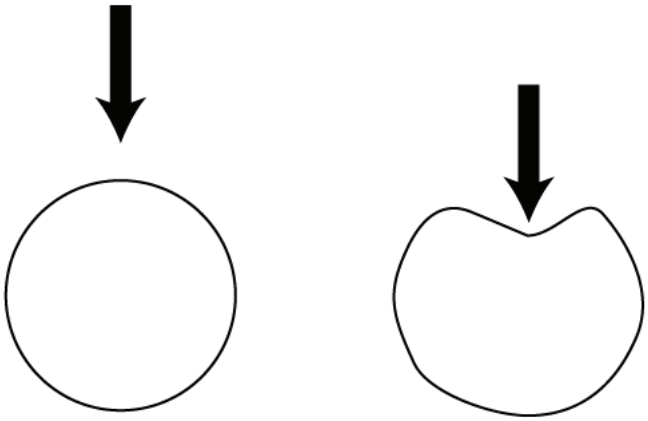


Figure 3.1: Distribution of applied force throughout the membrane of a balloon

structure. Compression members are discontinuous and their functionality is localised. They do not have to transmit loads over large distances and they are therefore are not subject to high buckling loads that would otherwise be the case. Hence these members can be made more slender without losing structural integrity. The tensegrity structures discussed in this dissertation are not frequently seen, but the structural principals of tensegrity are common in the surrounding natural and man-made environment. Pneumatic structures are tensegrity systems, for example, a balloon filled with air [9]. The membrane is the tensile component and the air inside is the compressive component. The membrane or skin of the balloon is made up of atoms and molecules linked to each other forming a continuous system. Conversely, the air molecules are highly discontinuous. If a force is applied to the balloon, e.g. pushing down on the skin, there is no cracking; the continuous, flexible netting formed by the balloon's skin distributes the force throughout the structure (Figure 3.1). Once the external force is removed, the structure returns to its original shape. This quality of resilience is an important characteristic of tensegrity systems. Prestressed concrete is another

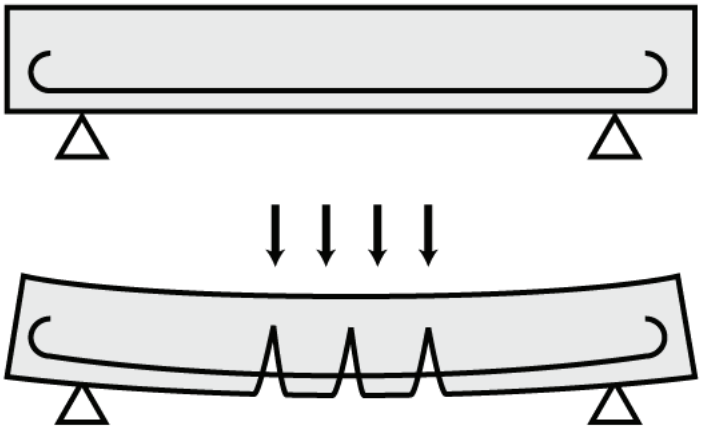


Figure 3.2: Pre-stressed concrete beam resisting both tension and compression.

example of a system that utilises tensegrity principles [43]. A pre-stressed concrete beam is composed of a concrete volume with internal steel tendons. These tendons are under high tension forces even without any external load applied to the beam. The concrete is consequently in compression (Figure 3.2). The tendons are positioned such that application of vertical load to the beam is resisted in tension, whereas the concrete, which is inefficient in tension, resists compression forces in other locations in the beam. The state of pre-stressed concrete, where forces are present even when there is no applied loading, is another distinguishing quality of tensegrity systems. In the natural environment, the structural framework of non-woody plants relies entirely on the tensegrity principle. Young plants are made up of cells of water which behaves similarly to the balloon described previously. The water inside the cell holds the skin of the cell in tension. This skin is a flexible membrane made up of an interconnected system of atoms. When the plant is subjected to forces and bent by the wind, rain and other natural forces, the forces are distributed throughout the plant without compromising the plant's structural integrity. Even after undergoing great distortion from its

original shape, once the forces are removed, the plant returns to its initial form. The presence of water in the plant is critical to its structural stability, and this usage is evident when the plant dries out and wilts.

3.2 Motivation

Structures are becoming lighter and this lightness has gained increased appeal by architects and engineers due to the aesthetic and structural advantages it provides [22]. The unlimited length and versatility of tension structures are gaining popularity as can be seen in construction . Combining tension structures with roof materials such as glass and membrane gives a sense of transparency and lightness in structural form. Innovation in forms is constantly occurring in order to achieve unique and aesthetically appealing structures that meet the imagination and curiosity of people. Tension structures are at the forefront of these new forms. Space structures themselves often express architectural art, with modern structural forms giving rise to new aesthetic standards. Tensegrity systems, as a form of tension structure, are included among the newly developed forms and are thus gaining popularity today. Since art, originality and personal expression are the major concerns for architects, they may not pay attention to structural form, rationality and scientific principles. Looking at tensegrity systems from a structural point of view leads one to the classification of tensegrity as a type of cable-strut system. Using this classification we can begin to study the design and analysis of these systems.

3.3 Benefits of Tensegrity Systems

Tensegrity systems have been extensively studied in terms of geometry, architecture and aesthetics, but there have been few comprehensive studies on the mechanics and dynamics of these systems [44]. Generally simple tensegrity

structures and modules are developed, analysed, and these have been used as building blocks to construct larger systems. The following is a list of the advantages tensegrity structures offer and why this type of structural system is receiving increased attention from engineers, architects and mathematicians.

3.3.1 Tension Stiffness

Tensile members exhibit increased stiffness under loading whereas compressive members have reduced stiffness. In compressive members, when loading is axial only, stiffness is lost due to the increase in area of the cross-section resulting from the spread of material in this region. When there exists bending moments in a compression member, possibly due to the eccentricities of the applied axial loads, the stiffness of the member is reduced due to this bending motion. Generally, the tensile strength of a member is larger than its compressive (buckling) strength (excluding materials like sand, masonry and unreinforced concrete). Therefore by using more tensile members, a larger stiffness-to-mass ratio can be attained.

3.3.2 Efficiency

Material layout and geometry is essential to structural strength. This applies to structures of any scale, from the nanoscale biological structures, to large scale civil buildings. Structures are traditionally built rectilinearly, utilising orthogonal beams, slabs and columns. It has been seen that orthogonal architecture does not result in the minimal mass configuration and design for a given stiffness property set. Neither a solid mass of material with fixed external geometry, nor orthogonally arranged material components have proven to be the optimal mass distribution for a given set of stiffness properties. Material is only required along critical load paths and not along the orthogonal paths of traditional built structures. Tensegrity systems are composed of non-orthogonal member configuration layouts which result in increased

strength with lower mass. Self-similar constructions are also used to increase the efficiency of tensegrity structures. This is done by replacing a tensegrity member with another tensegrity system.

3.3.3 Deployability

High strength materials generally do not allow large displacements. These materials are therefore only capable of small displacements. In tensegrity systems, the compressive members are disjointed and connected with ball joints. This results in the advantages of; allowing large displacements, deployability, compact storage and more convenient transportation. This essentially allows a tensegrity structure to be portable and allows certain operational advantages. A tensegrity bridge or tower can be fabricated and assembled in a workshop, transported to site and erected (deployed) by increasing the tension in various cables, either by cable tensioners or winches. This construction process can also be applied to temporary structures such as exhibition domes or shelters. In space applications, deployable tensegrity structures can be combined with active control systems to reduce launch costs by decreasing the mass or reducing human assembly requirements.

3.3.4 Shape-Shifting

"Tuning" of a structure is achieved by adjusting the length of certain members to achieve a shape or stiffness change. Fine tuning of a structure can be applied similarly to the deployable configuration technique. This system can be utilised for loaded or damaged structures.

3.3.5 Reliable Modelling

Every member of a tensegrity structure is axially loaded. The system as a whole can bend under the application of pre-stress forces or external loads, but the individual members of the system experience no bending moments.

Tensegrity systems are generally designed such that the members undergo loading that is below their Euler buckling load. It is harder to model members that experience forces in more than one direction. The Euler buckling load of a compressive member, which is calculated from a bending instability calculation, is known to be unreliable in practice. Tensile strength is more predictable than the actual buckling load, since the calculated buckling load measured from test data has a larger variation. Increased use of tensile members is expected to result in better theoretical models and therefore more efficient structures. It is thus noted that more reliable and accurate models can be developed for axially loaded members compared to that of bending members.

3.3.6 Control

Since tensegrity structures can be easily modeled, they can therefore be more precisely controlled compared to other structures. The configuration determines the structural and mathematical properties and these properties can be scaled from nanoscale to megascale.

3.3.7 Integration of Structure and Control Disciplines

Tensile and compressive members of tensegrity structures can be used for multiple functions simultaneously. These functions include; load-carrying member, sensor, actuator, thermal insulator or electrical conductor. This means that by carefully choosing the materials and geometry of a tensegrity structure, it can be designed to control the mechanical, electrical and thermal energy of the material and structure. An example of this would be a tensegrity aircraft wing where shape control is utilised to control and manoeuvre the aircraft or to optimise the aerofoil as a function of the flight conditions, all this without the use of hinged surfaces. Tensegrity structures offer promising opportunities for integration of structure and control design.

3.3.8 Motivated from Biology

The nanostructure of the spider fibre is a tensegrity system. Spider fibre is the strongest natural fibre per unit mass. Some argue that tensegrity is the fundamental building system of life. This observation is based on experiments in cell biology, where truss structures of pre-stressed tensegrity systems have been identified. Similarities to this have been observed at various scales in nature. Natural structures possess tremendous efficiency and if tensegrity systems are the building blocks of nature, then modern analytical and computational study of tensegrities could make these efficiencies possible for man-made structures in a wide range of applications and scales.

3.3.9 Small Storage Volume

Removal of key struts or releasing certain members can allow a tensegrity structure to collapse into a small volume for easy storage [9]. The structure can thus be contained in a small space for storage or for transport. It is also possible to disassemble the structure into parts for those same purposes. Erection at the desired location can then take place through assembly of parts. In the case case where particular cables or struts were removed, these members can then be reattached to deploy the system into its final configuration. Decreasing the volume of a structure for storage or transportation is of particular importance in fields such as space applications, aquaculture installations and for temporary structures [45].

3.4 Definition

Controversy still exists over the origin of the tensegrity concept. There is no universally accepted definition for these systems, and previous attempts at providing a completely unambiguous definition has proven difficult. The concept can be illustrated by quoting Fuller as he describes the tensegrity

principle as "islands of compression inside an ocean of tension." Looking at this broad definition, it is clear that many objects could be related to this principle. For example, a balloon or any inflated envelope can fall under this definition since no specifics on material or form is mentioned in the principle. However, suggestions are given on the segregated property of compression members in the tension members' configuration. The compressed air in a balloon is held inside a tensioned membrane. Establishing a clear definition on tensegrity systems has proven difficult. Definitions based on patents provide some clarity on the concept and serves as a reference for comparison against other known definitions. Patents submitted by Fuller, Snelson and Emmerich all describe the same structural system, and in this meaning the following patent-based definition is established [9]:

Tensegrity Systems are spatial reticulate systems in a state of self-stress. All their elements have a straight middle fibre and are of equivalent size. Tensioned elements have no rigidity in compression and constitute a continuous set. Compressed elements constitute a discontinuous set. Each node receives one and only one compressed element.

This definition, although descriptive, is still fairly generalised and a more suitable technical definition is required.

3.4.1 Structural Definition

Tensegrity systems are free-standing pin-jointed networks in which an interconnected system of cables are stressed against a disconnected system of struts or extensively, any free-standing systems composed of tensegrity units satisfying the aforesaid definition [22].

This definition distinguishes tensegrity systems from conventional cable networks. Cable networks achieve equilibrium of the entire structure by means of anchorage systems and pre-stress. Cable domes and spoke-wheel domes are however derived from the tensegrity principal. These two types of domes utilise continuous tensioned cables and discontinuous compressive

posts to span the enclosed space. Sometimes these types of structures may be mistaken as tensegrity structures, but their mechanical properties are significantly different and therefore they do not fall into this category.

In these systems, all connections between components are pin jointed and all members are only subjected to axial forces. Components include bars that can resist both compression and tension, and cables that can only resist tension. For strictly defined tensegrity systems, bars are reduced to struts that can only resist compression. A large variety of space structures can be categorised as pin-jointed systems, for example space trusses and cable suspended structures. It is therefore clear that tensegrity systems are a type of pin-jointed system. Classification of pin-jointed systems is characterised by independent self-stress states and independent mechanisms. The classification is based on the initial geometry of the system and the mechanisms may be finite mechanisms or infinitesimal mechanisms. Infinitesimal mechanisms disappear when the displacement is less than the length order, whereas finite mechanisms would still exist. A system containing infinitesimal mechanisms has at least one self-stress state.

3.5 Properties of Tensegrity

Stability is an important aspect of tensegrity systems. The structural, architectural and mechanical properties result from the continuity of the tensioned components and from the discontinuity of compressed components [9]. Theoretical bases, such as graph theory, together with the concept of relational structure, clearly characterise these continuity and discontinuity concepts. States of self-stress and the stabilisation of infinitesimal mechanisms are the underlying requirements for stability and stiffness of tensegrity systems. Only geometry that is governed by static equilibrium conditions allows this stiffening to be possible. It is therefore important to explain the concepts of finite and infinitesimal mechanisms, as well as associated states of self stress as

they relate to tensegrity systems. Stabilisation of infinitesimal mechanisms is a critical subsystem and it reveals the reasons why tension and compression elements cannot be exchanged by simple duality.

3.5.1 Relational Structure

This is defined as a reticulate system being created by an assembly of components. The components may themselves be comprised of several subcomponents.

3.5.2 Geometry and Stability

A spatial reticulate system is defined completely by its relational structure and by the values of the x, y and z co-ordinates of the n nodes with reference to a chosen co-ordinate axis system. Stability of a tensegrity system can only be satisfied for a geometric configuration that is in stable static self-equilibrium. To determine this equilibrium state, a form-finding process is required.

3.6 Structural Aspects

3.6.1 Application of tensegrity

The appeal in the use of tensegrity structures lies in their quality of resilience and their efficient and economic use of materials [43]. They make effective use of the advantages of tensile performance and advances in engineering relating to material technology. Tensegrity structures predominantly use tensile members while the use of bulky compression members is minimised. The use of tensegrity design in construction of buildings, bridges, towers and other structures allows the structure to be highly resilient and more economical than conventional structural designs.

3.6.2 Typologies

This dissertation deals only with tensegrity systems that can be defined as spatial reticulate systems. All tensegrity modules, assemblies and structures investigated in the following chapters comply with the structural definition given in Section 3.4.1. Other tensegrity systems such as biological cells are therefore excluded. There are many factors that determine the typology of a system; topology, geometry, mechanical characteristics [9] (self-stress states and infinitesimal mechanisms) and others. Three types of tensegrity systems have been developed, namely cellular or elementary units, their assemblies, and complex systems without an identified constitutive cellular unit.

3.6.3 Pre-stress versus Stiffness

Increasing pre-tension of members results in increased robustness to uncertain disturbances. Generally when a load is applied to a tensegrity structure, the stiffness does not decrease, unless one or more cables go slack. It has been shown [44] that the effect of pre-stress on the stiffness of a tensegrity without slack cables is almost negligible. Also, the bending stiffness of a tensegrity without slack cables is not significantly affected by pre-stressing.

3.6.4 Small Control-Energy

The shape of a tensegrity structure can be changed with only a small amount of energy being used by the control mechanism [44]. This can be achieved because shape changes are achieved by reconfiguring the structure to a new equilibrium position. Thus energy is not required by the control system to hold the new configuration. This is in contrast to traditional structures which require energy to keep the system in a different configuration to the initial equilibrium.

3.7 Design

It is important to define the main aspects that form the basis of tensegrity system design. Also to highlight the resulting mechanical and structural problems associated with these aspects.

The critical aspect is the initial state of the system and its behaviour when subjected to external disturbances. The initial state of the system is important since it is a self-equilibriated state. Additionally, all components have unilateral rigidity (either in compression or in tension), and the relational structure is explicit; discontinuous compression components are in a continuous set of tension components.

Every system with initial stresses can be defined by two sets of parameters; form and force parameters. Form is established based on the relational structure and the geometric properties of the manufactured components. Form can be easily quantified as it can be seen. Force parameters rely on the capability to introduce an initial stress state in the system to assure its stiffness. This stress state however cannot be seen directly. It is therefore favoured to apply deformations to the components of the system to produce the stress state.

Key challenges to solve in the design of tensegrity systems include [9]:

- Form-finding
- Self-stress feasibility
- Compatibility between self-stress and component stiffness
- Identification of mechanisms
- Stabilisation of mechanisms
- Sizing of components
- Mechanical behaviour under external actions

- Sensitivity to imperfections

3.7.1 Form-finding

This is an essential problem in the design of tensegrity systems, since the fulfilment of stability requirements depends on both the shape and geometry. The solution requires simultaneously solving the geometry and self-stress. Thus any form-finding method would be either based on geometry or mechanics, but both aspects still need to be taken into account. Motro [9] claims that two main methods are available, namely form-controlled and force-controlled. The first method was used extensively by sculptors, specifically Kenneth Snelson [9]. The method works by developing tensegrity forms without any regularity of components or generalisation of mechanical properties. A heuristic method based on experimentation and a trial-and-error process is used to achieve stability. This method has frequently produced notable results. Force-controlled methods use theoretical models to ensure that the mechanical requirements for stability are met. It is clear that these methods need to simultaneously take into account both the geometry and the pre-stressability in order to achieve stability. These design techniques can give accurate results yet they can as easily fail. If the results of this method are highly regular systems, then the heuristic methods offer greater advantages based on this criteria. It should also be noted that mechanically based solutions require longer development times for complex systems.

3.7.2 Self-stress and Mechanisms

Another essential problem in tensegrity system design is rigidity, which is governed primarily by self-stress. Tensegrity systems usually have infinitesimal mechanisms, and may, or may not, be stabilised by the self-stress states.

3.8 Analysis

There are two steps in the analysis of cable net type structures [22]. Firstly the form-finding (shape-finding) process for establishing the equilibrium geometry when self-stress is applied. Secondly is studying the behaviour of the system under external actions, i.e. analysis of load response. Methods based on stiffness by themselves cannot be used to analyse structures with finite mechanisms or higher order infinitesimal mechanisms. Nor can they be used exclusively for the form-finding process when infinitesimal mechanisms exist, or for analysis under unstable loading resulting in the slacking of cables. However, these problems may be avoided by introducing dumb components that model the structure as a system with mechanisms that are equivalent to the 'geometrically rigid' structure.

3.9 Other

3.9.1 Foldable Tensegrities

The usage of folding principles to adapt tensegrity systems into foldable structures has significantly increased the field of applications for tensegrity.

3.9.2 Tensegrity as a structural principal

Even though tensegrity systems are not extensively used in the fields of architecture and engineering, it is clear that the interest and study into these systems will only keep increasing in the near future [9]. Tensegrity concepts have attracted increased attention from other fields, specifically biology. Both the challenge and excitement in the field of tensegrity is the selection of materials and geometry to manipulate thermal, electrical, mechanical and structural aspects and to integrate these disciplines in an efficient control system.

Chapter 4

Form-Finding by the Dynamic Relaxation Method

Several techniques exist for the form-finding and static analysis of tensegrity systems. Many of these techniques have been reviewed in Section 2.2 to outline the work previous studies have done to solve the tensegrity design problem. This dissertation uses the dynamic relaxation method, introduced in Section 2.2.6, for design applications. Here we give more details.

The dynamic relaxation method is based on a trace of the motion of every node in a structure, step by step for small time increments, until it relaxes into static equilibrium due to artificial damping [31]. For form-finding, the process may be initiated from an arbitrary geometric configuration, starting the motion by applying forces or stresses to nodes or elements in the system. Analysis of the structure must start from a correct initial or pre-stress equilibrium state, with the motion being started by applying the loading [31].

The dynamic relaxation method is an attractive method for the design of cable networks [46], and similarly tensegrity systems, because both form-finding and analysis can be resolved within the same procedural framework.

4.1 Advantages

Unlike other numerical formulations, the dynamic relaxation method does not rely on the assembly of any matrices. This means that complex matrix manipulations are not required for the solution of the system of non-linear equilibrium equations [47]. It is widely recognised as a successful method in the modelling of prestressed cable nets and membranes.

4.2 Formulation

Under static equilibrium, a system of N equations must satisfy [47]:

$$\boldsymbol{F}_i = \boldsymbol{F}_e \tag{4.1}$$

Where $\boldsymbol{F}_i$ is the internal force vector which is a function of the systems current configuration $\boldsymbol{x}$, and $\boldsymbol{F}_e$ is the applied static external force vector acting on the structure.

The equation of motion for a discretised system is [47]:

$$\boldsymbol{F}_e = \boldsymbol{M}\,\ddot{\boldsymbol{u}} + C\dot{\boldsymbol{u}} + \left[\sum K\boldsymbol{u}\right] \tag{4.2}$$

Where $\boldsymbol{F}_e$ is, as previously, the vector of external forces, $\boldsymbol{M}$ represents the lumped nodal masses, C is the viscous damping coefficient, K is the stiffness and $\ddot{\boldsymbol{u}}, \dot{\boldsymbol{u}}, \boldsymbol{u}$ are the vectors of nodal acceleration, velocities and displacements, respectively. The vector of internal loads, $\boldsymbol{F}_i$, is therefore $\sum K\boldsymbol{u}$. The nodal residual forces, $\boldsymbol{R}$, is defined as the difference between the internal and external force vectors.

$$\boldsymbol{R} \;=\; \boldsymbol{F}_e - \left[\sum K\boldsymbol{u}\right] \tag{4.3}$$

$$= \boldsymbol{F}_e - \boldsymbol{F}_i$$

Therefore:

$$\boldsymbol{R} = \boldsymbol{M}\,\ddot{\boldsymbol{u}} + C\dot{\boldsymbol{u}} \tag{4.4}$$

This states that the motion of the system is due to the unbalanced forces. The requirement for static equilibrium, based on Equation 4.3 is that the unbalanced forces must be zero.

Equation 4.4 can be approximated by centred finite difference methods. This is achieved by calculating the acceleration from the initial and final velocities over the time interval $\triangle t$. The velocity is taken as the average velocity over the same interval. Let n be the time increment at which variables are calculated[47], then:

$$\boldsymbol{R}^n = \boldsymbol{M}\frac{\dot{\boldsymbol{u}}^{n+\frac{1}{2}} - \dot{\boldsymbol{u}}^{n-\frac{1}{2}}}{\triangle t} + C\frac{\dot{\boldsymbol{u}}^{n+\frac{1}{2}} + \dot{\boldsymbol{u}}^{n-\frac{1}{2}}}{2} \tag{4.5}$$

From this equation (4.5), follows the recurrence equation for velocities:

$$\dot{\boldsymbol{u}}^{n+\frac{1}{2}} = \left\{\dot{\boldsymbol{u}}^{n-\frac{1}{2}}\frac{\frac{\boldsymbol{M}}{\triangle t} - \frac{C}{2}}{\frac{\boldsymbol{M}}{\triangle t} + \frac{C}{2}}\right\} + \frac{\boldsymbol{R}}{\frac{\boldsymbol{M}}{\triangle t} + \frac{C}{2}} \tag{4.6}$$

These velocities are used to estimate the displacements at time $n+1$:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \dot{\boldsymbol{u}}^{n+\frac{1}{2}}\triangle t \tag{4.7}$$

The overall solution involves an iterative process using Equations 4.6 and 4.7 repetitively. The process is repeated until the residuals are equal to zero or are sufficiently close. The residual forces are calculated using Equation 4.8, which in the iterative process is:

$$\boldsymbol{R}^n = \boldsymbol{F}_e - \left[\sum K\boldsymbol{u}\right]^{n-1} \tag{4.8}$$

$$= \boldsymbol{F}_e - \boldsymbol{F}_i^{n-1}$$

4.2.1 Convergence

Originally, the dynamic relaxation procedure was applied with viscous damping, obtaining convergence of the system by critically damping the lowest form of vibration. Additional controls were sometimes necessary to achieve convergence. This was required in instances of very large locally unbalanced forces, sometimes occurring due to excessive deformation of members during the initial form-finding steps of very inaccurate starting geometry [31]. To cope with this problem, the kinetic damping procedure was found to be stable and convergent (See Section 4.2.2). This damping procedure monitors the total kinetic energy of the system, and when a local peak is detected, all the nodal velocity components are set to zero. The system is then restarted from the current geometry (at which the peak occurred) and the process is repeated through further (generally decreasing peaks) until the energy of the entire system for all modes of vibration is dissipated . Under zero kinetic energy, the system is in a state of static equilibrium.

4.2.2 Kinetic Damping

The dynamic relaxation technique used throughout this dissertation uses the kinetic damping procedure. It is formulated by taking the viscous damping coefficient in Equation 4.4 equal to zero:

$$\boldsymbol{R} = \boldsymbol{M}\ddot{\boldsymbol{u}}$$

This results in the following recurrence equation for velocity:

$$\dot{\boldsymbol{u}}^{n+\frac{1}{2}} = \dot{\boldsymbol{u}}^{n-\frac{1}{2}} + \frac{\boldsymbol{R}\triangle t}{\boldsymbol{M}}$$

To determine the updated geometry, displacements are calculated using:

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^{n} + \dot{\boldsymbol{u}}^{n+\frac{1}{2}}\triangle t$$

4.2.3 Stability of the iterative solution

If the time increment $\triangle t$ is too large or the mass too small, then the iterations will become unstable and convergence will not occur. For the the iterative solution to be numerically stable [31], the time increment must comply with:

$$\triangle t \leq \sqrt{\frac{2M}{K}}$$

For an arbitrary time increment $\triangle t$, the mass is calculated from:

$$M_i = \lambda \frac{\triangle t^2}{2} K_i \tag{4.9}$$

where M_i is the lumped mass at node i, K_i is the stiffness, λ is a convergence parameter which is constant for the whole structure. The value of M_i can be chosen at convenience to optimise convergence, since the objective of the process is not to trace the real dynamic behaviour. The stiffness K_i is taken as the maximum K of the principal directions (See Section 4.2.4). The convergence parameter λ was proposed by Han and Lee [48] as a way to adjust the system to achieve the optimal mass for convergence of the solution. Lumped nodal masses can be used to reduce computation time.

4.2.4 Stiffness

To calculate the stiffness K in the above equations, two types of stiffness terms need to be taken into account; elastic stiffness and geometric stiffness. The elastic stiffness of a member is dependent only on the material properties of that element, whereas the geometric stiffness term is a function of the geometry, displacement, and state of stress of the element.

Elastic stiffness:

$$K_e = \frac{EA}{L_0}$$

Where EA is the axial rigidity, L_0 represents the original, undeformed length of the member.

Geometric stiffness

$$K_g = \frac{T}{L_c}$$

Where T is the internal force in the member, due to pre-stress or stress resulting from deformation due to applied loading. $L_c$represents the current length, after deformation of the member has occurred.

In Equation 4.9, K_i is the greatest direct stiffness of the local principal directions at each node [31]. This implies that only one lumped mass value related to each node is considered.

4.2.5 Member and Nodal Forces

The internal force in any member [48] is calculated from:

$$T_m^n = \frac{EA}{L_m^0}(L_m^n - L_m^0) + T_m^0 \tag{4.10}$$

Where T_m^n is the internal force in member m at time increment n (current time), L_m^0 and L_m^n represents the initial and current lengths of members

respectively, and T_m^0 is the initial prestress introduced in the member. From Equation 4.10, the nodal force can be calculated:

$$F_{ij} = \sum_{m=1}^{N_i} \frac{T_m^n}{L_m^n} \left\{ (X_i^\alpha)^n - \left(X_i^\beta \right)^n \right\}$$

Where F_{ij} is the internal force at node i in the j direction, N_i is the number of members at node i, α and β are the start and end positions of member number m respectively.

4.3 Iterative Procedure

The iterative process [31] can be summarised as follows (Figure 4.1):

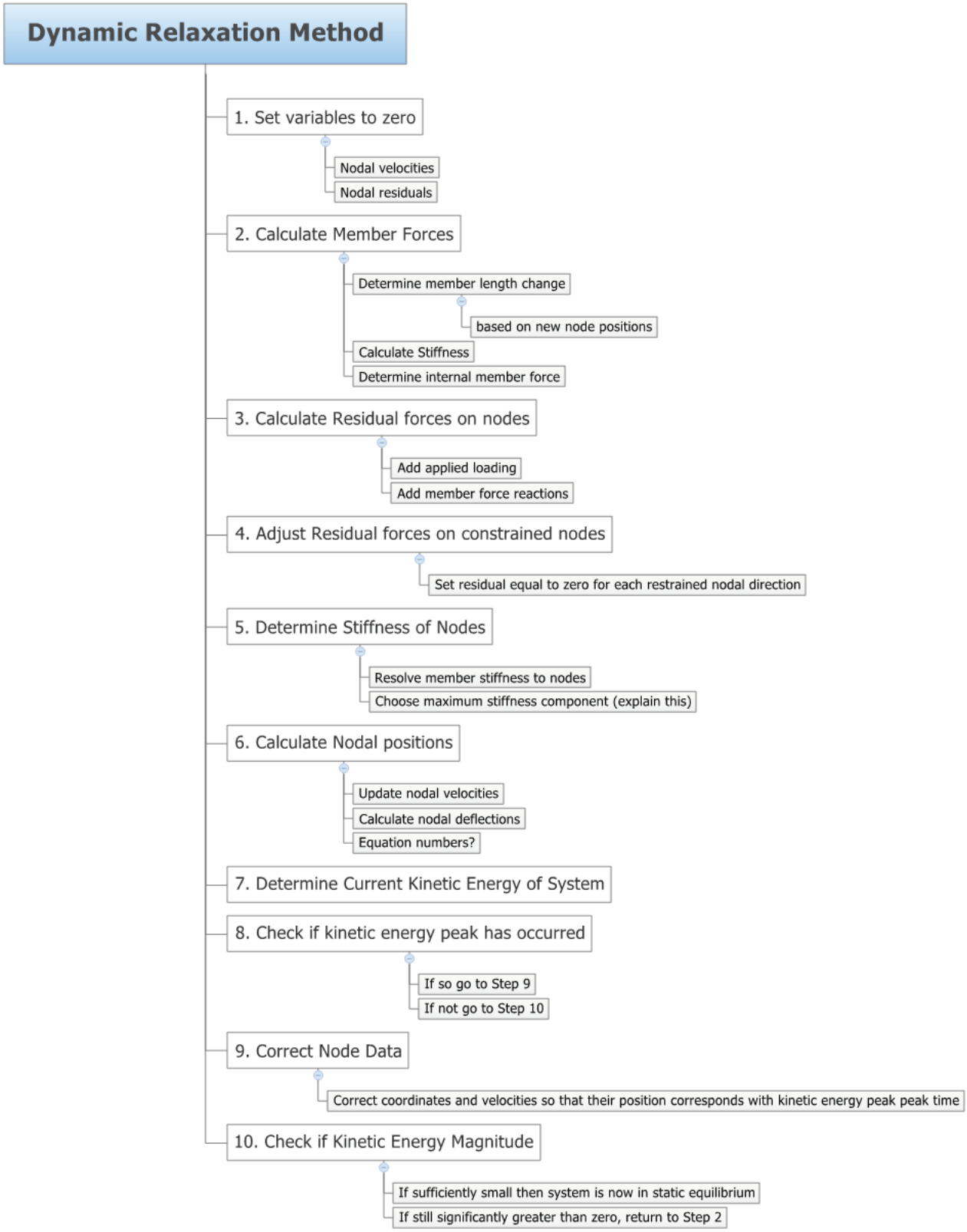


Figure 4.1: Flow chart showing iterative procedure of the dynamic relaxation method

Chapter 5

Tensegrity Design Program

In order to obtain the solution from the dynamic relaxation method described in the previous chapter, a software application is now developed. The mathematical iterative procedure is programmed in a Visual C# application called Tensegrity Designer (Figure 5.1). This piece of software was developed especially for this thesis by myself. The full program source code listing can be found in Appendix A.

The program's algorithm and its mathematical principles were described (and sources cited) in Chapter 4. I have taken this mathematical formulation and adapted it in order to develop an iterative computational procedure which I used to program a custom software solution. I then further developed this software and added a graphical user interface to allow programmatic modelling of tensegrity structures and visualisation of the results.

The aim of Tensegrity Designer is to calculate the equilibrium position of any tensegrity, given the topology and loading. Thus given the initial topology, the initial equilibrium could be calculated, similarly, by applying loads to the calculated equilibrium position, the deflections and member forces can be determined for various load cases.

Development of this application is needed since no publicly available, general-purpose static analysis or finite element software specifically allows

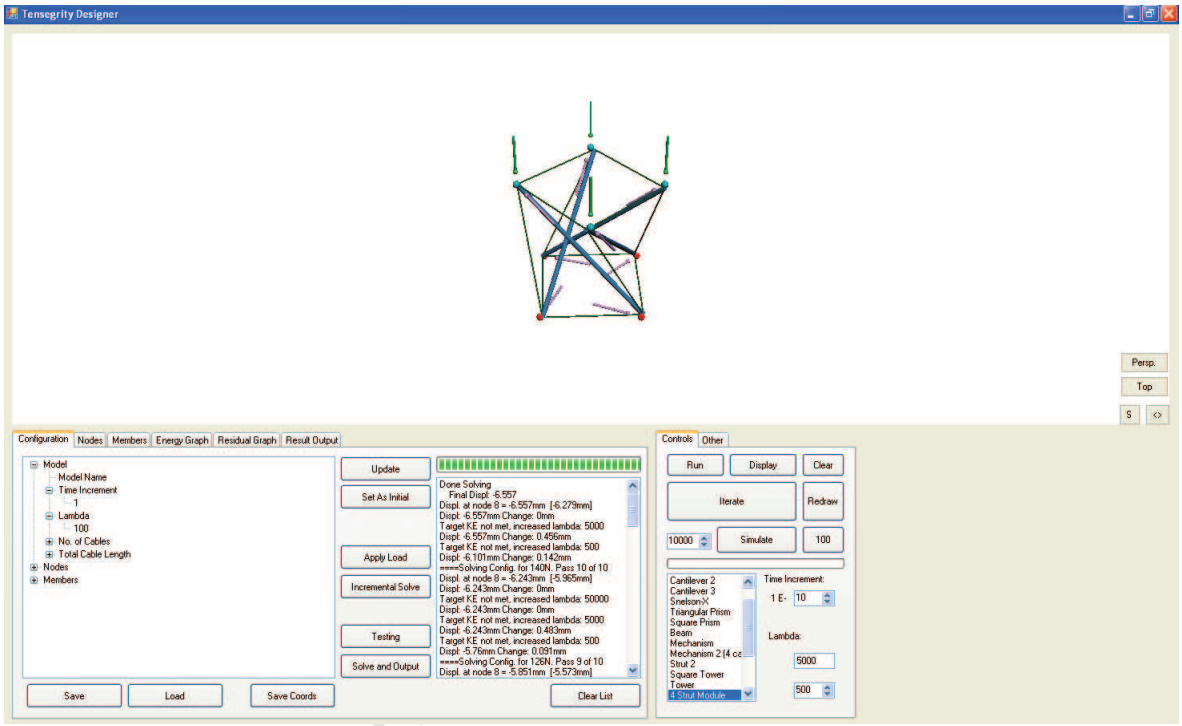


Figure 5.1: Screenshot of Tensegrity Designer

for form-finding of tensegrity systems and its associated specialised requirements. Certain researchers have, in their personal capacity or with their research groups, developed tensegrity design software, but these are not generalised enough, or not freely available. It proved more useful to develop the software using the mathematical model developed. This allowed complete control and customisation of the program to suit the design requirements.

The structural system is modelled as a set of nodes, interconnected via cable or strut elements. Masses, stiffnesses and forces are lumped and nodes and calculations of residual forces are carried out. These forces result in nodal movements, which are traced step by step over defined time increments. Local coordinates are used as parameters and forces are calculated at each node. Every iteration of the procedure cycles through every element and resolves its stiffness and force properties to its connected nodes. Each node is then considered and residual forces calculated, as well as velocity and then displacements. Calculation of nodal displacements gives the updated geometry of the system, independent of rigid body motion which is treated automatically by the dynamic relaxation process [49].

5.1 Program Logic

The program is developed using classes. Each class represents an object in a tensegrity structure or a substantial part of the program. In programming terms, a class is an "object" containing "properties" and "methods." Properties are variables or values and methods are functions that can be called to run calculations or perform tasks.

5.1.1 Vector Class

This class is used to simplify vector calculations for the other classes. It represents vectors by storing the X, Y and Z components in its properties. It allows calculations and manipulations to be performed on and between vec-

tors, namely; calculation of the dot-product, cross-product, vector addition, multiplication, creating unit vectors, finding perpendicular vectors, and resolving vectors along other vectors, such as when calculating member forces. The Vector class forms the foundation on which more complex classes will be built upon.

5.1.2 Node Class

The Node class represents a node in the tensegrity system. This is the point at which members meet and forms the joints in the structure. The main properties of this object are the 3-dimensional Cartesian coordinates, vector of applied forces, stiffness vector and mass. This class also defines the boundary conditions of the node, i.e. which directions allow free movement and which are fixed. The dynamic relaxation method requires all masses, forces and stiffnesses to be lumped at nodes, from which it calculates the residual force vector, nodal velocity and the new position of the node after the set time increment. These calculations are all computed by methods coded in the Node class.

5.1.3 Member Class

The Member Class defines each member in the system. This definition is based on the start node, end node, member type, material properties, internal forces, geometric properties and orientation in the system.

5.1.4 MemberType Class

This class describes the type of member represented by the Member class and the properties associated with it. It defines whether the member is a compressive or tensile member, the geometric properties such as radius and cross-sectional area, and the material properties including Young's modulus

and density. It contains methods for computing both the elastic and geometric stiffness of the member, its mass, as well as methods for distributing these properties onto its connected nodes.

5.1.5 Tensegrity Class

This is the main class and manages the running of the application. It stores all the nodes and members in arrays of Node and Member classes, respectively. The Tensegrity class represents the entire tensegrity model.

5.2 System Geometry Definition

Geometry is input by first defining nodes using the Node class, including properties such as position and nodal fixity. In order to create members in the system, the basic member types have to defined by MemberType classes, i.e. strut and cable members have to be defined and the material properties of each is to be established. Members can then be added by specifying their start node, end node, and type. During this initial geometry creation, nodal loads and member pre-stress forces can be defined. Other input parameters that are required, include the time increment between iterations.

5.3 Computation of the Iterative Solution

The first step in establishing a solution is to calculate the nodal values after one iteration. The Tensegrity class calls its *Run()* method which cycles through all of the members and calculates the properties of each before resolving them onto their associated nodes. It then cycles through each node and, using the properties calculated (mass, forces applied and stiffness), calculates the nodal velocity and displacement during the time increment. Once all the displacements are determined, the new configuration of the tensegrity system at the current time increment is established. Kinetic damping

is used to determine the equilibrium configuration of the system. Thus the total kinetic energy of the system is calculated after each time step and monitored. This value is compared with values computed at previous time steps. If a kinetic peak has occurred, the time position of this peak is calculated and the new displacements for this time position is computed. The system's configuration is now set to that of the kinetic energy peak. At this point, all nodal velocities are set to zero, and the iterative procedure continues until the kinetic energy is sufficiently close to zero, at which point the system is taken to be in equilibrium. The kinetic energy peaks are generally decreasing and represent successively improving solutions to the overall equilibrium problem [49].

In order to optimise convergence of the solution and to avoid instability of the iterations, the λ parameter can be adjusted when required. Initially set as unity, this parameter controls the magnitude of the residual forces and thus the size of displacements. Increasing the value of λ decreases the relative displacements and avoids the large initial deformations during form-finding and analysis.

5.4 Visualisation

In order to visualise the tensegrity system and also to observe the configuration at each time step, a 3-dimensional visualisation "control" for Visual C# was developed (Figure 5.2). This control is a view of the graphical representation in 3-dimensional space. The structure can be rotated about any axis and zoomed as required. This is an important feature of Tensegrity Designer, as it allows visual confirmation of the configuration changes after each time step including the overall system after form-finding and analysis.

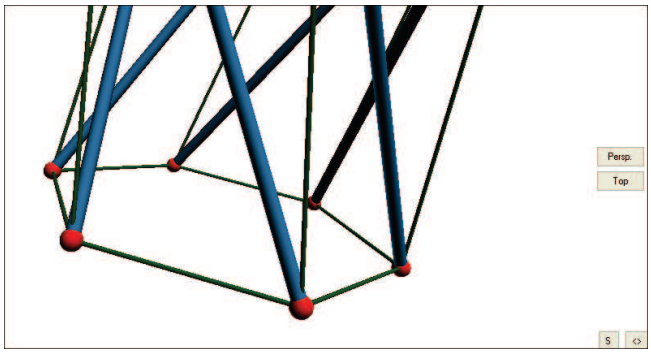


Figure 5.2: 3-Dimensional visualisation control

5.5 Data Persistence

Tensegrity Designer is able to store entire models to file for later use, including geometry, node positions, members details, connectivity information, forces, internal forces, as well as boundary conditions of nodes. It is also able to output the kinetic energy data at each time increment. This is stored in a file format that can be imported into any graphic visualisation software for graphing and evaluation of the employed damping system (Figure 5.3). Perspective views of models (from the view of the 3D visualisation control) can be exported to image files for usage elsewhere.

5.6 Summary of Form-Finding and Analysis Process

The process is summarised in the flow diagram shown in Figure 5.4.

5.7 Limitations

The program cannot create new forms, it can determine the stable self-stressed equilibrium configuration of a specified topology. If the initial con-

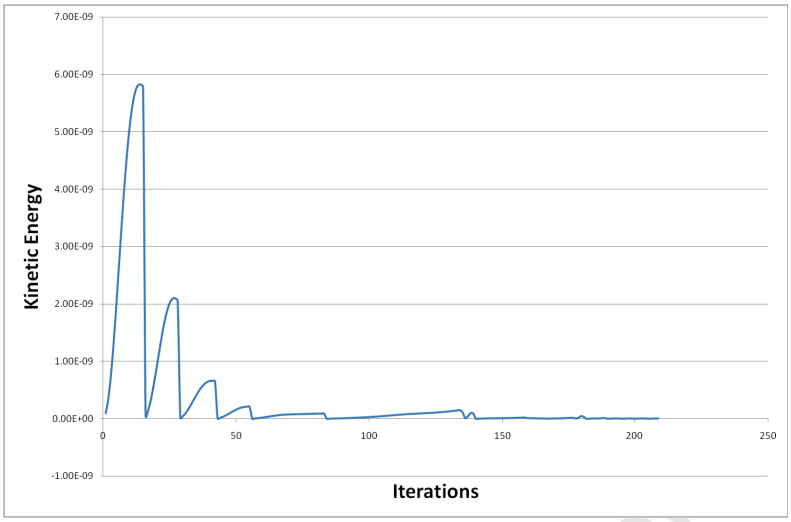


Figure 5.3: Kinetic energy dissipation over fictitious time (iterations) due to kinetic damping

figuration specified cannot be stabilised via pre-stress, then the solution will not converge and the kinetic energy of the system will not reduce to zero. Thus it is important for a reasonable initial configuration to be specified. However, defining a system that would simply collapse under the applied loading would result in rapid divergence and this would be clearly visible. The user can then adjust the input variables, whether by adding members or varying pre-tension forces, to produce a system that would result in a stable equilibrium configuration.

5.8 Possible Improvements

In theory it would be possible to use Tensegrity Designer to find equilibrium configurations by fixing the length of one member type and varying the other to induce pre-stress. This method was applied by Motro [30] when he fixed strut lengths and continuously varied cable lengths until stable equilibrium was achieved. This could be applied to Tensegrity Designer by defining infi-

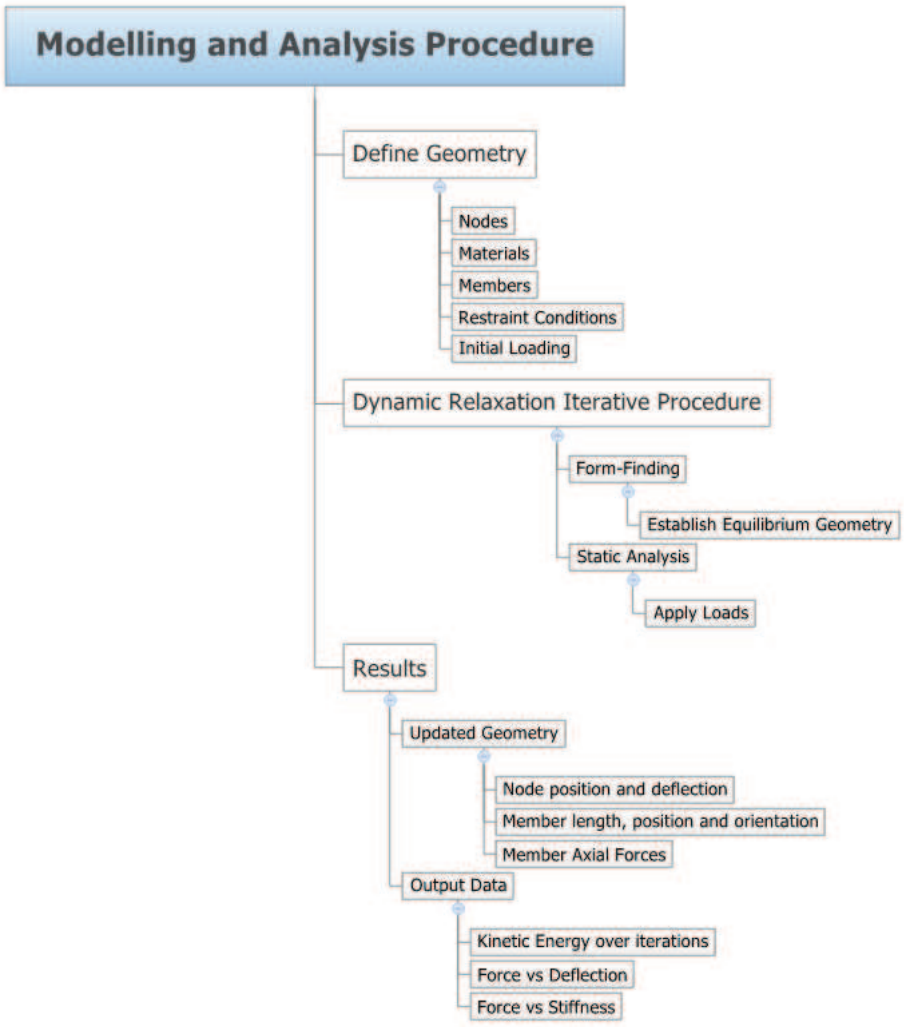


Figure 5.4: Flow diagram of Tensegrity Designer modelling procedure

nite stiffness for struts during the form-finding process. The result would be zero deflections in the directions of members and thus zero change member length. Although this was not the method used in the investigation undertaken, it could be easily applied by defining a MemberType with an infinite stiffness.

5.9 Usage

The models described in Chapter 7 were modelled in Tensegrity Designer using the aforementioned modelling procedure. It was initially noted that the effects of self-weight was insignificant to the overall system behaviour. Thus the self-weight load was ignored for all systems.

Once the system geometry had been completely defined, the loads were modelled according to the load cases described in Chapter 7. For each load case, convergence was optimised when required using the λ parameter discussed previously. Loads were applied incrementally to determine the deformation and stiffness characteristics of the system under the specific loading condition. Once the static equilibrium form for the particular load case was determined, the output data was saved. The results of these models can be seen in Chapter 8.

Chapter 6

Validation of Tensegrity Design Program

The tensegrity designer software was developed to aid in the design and analysis of tensegrity systems. In order to validate the program's algorithm and its results, the output was compared to theoretical models, and to results obtained from ABAQUS Finite Element Modelling software.

6.1 Infinitesimal Mechanism

Tensegrity structures are statically indeterminate structures, i.e. their static equilibrium equations are not adequate enough to determine unique internal and reaction forces. The geometry of the system is therefore an important aspect in the analysis of these systems. This is illustrated by Estrada [8] in Figure 6.1, where it is shown that the geometry and pre-stress greatly influence the statical determinacy or indeterminacy of the system.

In Figure 6.1a the centre node can resist vertical load P by transmitting compressive axial forces through the bars. The system will equilibriate the load and will be in equilibrium up to a certain level, at which point the phenomenon of "snap-through" will occur. This is a statically determinate

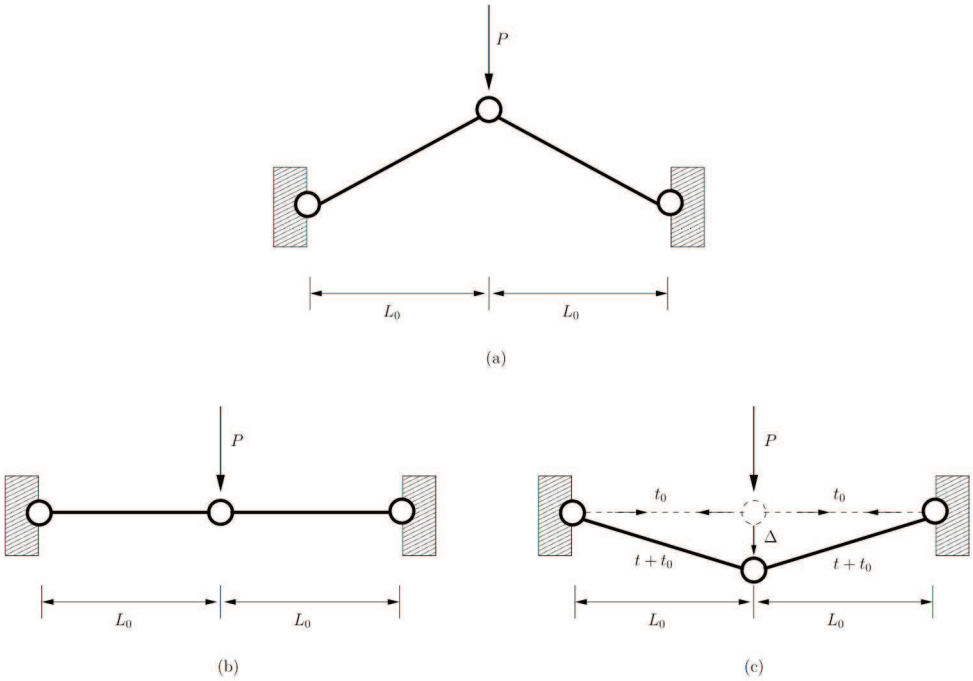


Figure 6.1: Deformation of a two-bar planar structure [8]

system.

The arrangement in Figure 6.1b, without any pre-stress in the bars, has no resistance to vertical loads. This is known as a finite mechanism. If however there is pre-stress present in the bars, as in Figure 6.1c, where the bars are under tension, i.e. $t_0 > 0$, then the system has an initial resistance against vertical loads. If we note that the bars behave as springs, then the deflection $\triangle$ due to P is a function of t_0. For this example this relationship is given as [46]:

$$\frac{EA}{L_0}\triangle^3 + 2t_0 L_0 \triangle - PL_0^2 = 0 \tag{6.1}$$

Here EA is the usual axial stiffness of the bars. Rearranged in terms of the load, P:

$$P = 2\frac{\triangle t_0}{L_0} + \frac{EA\triangle^3}{L_0^3} \tag{6.2}$$

Thus P is a third order function of the displacement $\triangle$ and the resistance to the load is proportional to t_0 as in the example in Figure 6.1c. In this case the system is stiffened by the pre-stress and this is known as an infinitesimal mechanism. The deformation mechanism is shown as solid lines in Figure 6.1c. The system is self-equilibriated, i.e. it does not require any external loads for equilibrium and could exist in a state of self-stress for $P = 0$, shown with the dotted lines in Figure 6.1c. The initial resistance of the system to vertical loading is a result of the pre-stress and not due to the axial stiffness, EA, of the bars. It is therefore said that the overall stiffness or flexibility of the structure is not dependent on the axial stiffness of the elements, but rather on the geometry of the system. This load carrying mode is called geometric stiffness [8]. Tensegrity can be characterised by their exhibition of geometric stiffness and stability induced by pre-stress.

The tensegrity design program was validated against the above equations. When there is no pre-stress in the bars, i.e. $t_0 = 0$, then equation 6.2 reduces to:

$$P = \frac{EA\triangle^3}{L_0^3} \tag{6.3}$$

A model was created in Tensegrity Designer with the geometric configuration as shown in Figure 6.1b, i.e. with no pre-stress present in the bars, and the following parameters:

- Cables used for the bars
- Cable material properties:
  - Diameter: 1 mm
  - Length: 0.3 m
  - Young's Modulus: $E = 193$ GPa

A load of P =1000 N was applied at the centre node in 10% increments, and the displacement $\triangle$ was calculated by the program using the dynamic relaxation technique. These displacement values were then used to calculate the theoretical P using Equation 6.3. An identical system was modelled in ABAQUS CAE and run to determine the deflection response due to the applied load. The centre node of this model was given an infinitesimally small initial displacement $\triangle$. This subsequently induces axial strain and therefore pre-stress in the members. Pre-stressing is required since ABAQUS cannot apply loading to a system with zero stiffness as this results in a *zero pivot* error. The small applied displacement converts the finite mechanism of the model (which has zero stiffness on its centre node) into an infinitesimal mechanism (which can resist forces due to axial stiffness components of the members in the vertical direction).

Figure 6.2 shows the differences between the results of the theoretical model, the tensegrity designer program results and the results of the ABAQUS model. The results show good agreement and are practically identical for all three models. In the displacement range between 0 and 0.03 m, there is some

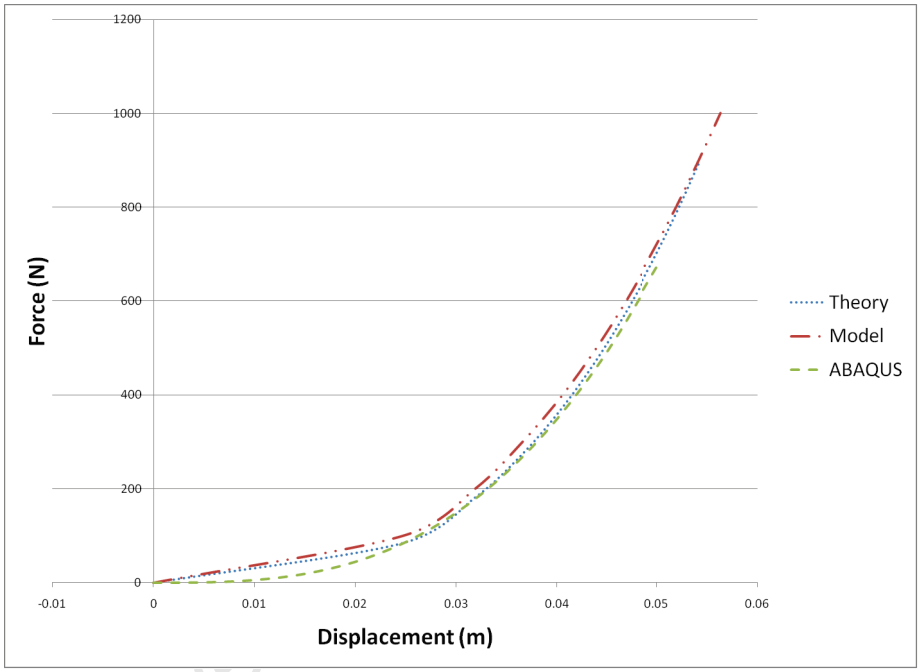


Figure 6.2: Force vs Deflection curve for two-bar mechanism with no pre-stress

variation between the ABAQUS results and those from the numerical model and theoretical model. This can be attributed to the initial displacement applied on the middle node in ABAQUS to stabilise the system and induce infinitesimal mechanisms.

6.2 Four-Strut Module

This design comprises four struts held in position via a system of interconnected cables (Figure 6.3). This basic module is based on the half-cuboctahedron presented by Quirant et al. [50]. It is also classified as a diamond tensegrity by Pugh [10]. The base cables are arranged in a square and each strut is connected to a corner. Struts extend upwards and diagonally across, leaning inwards. The top ends of the struts are joined via cables arranged in a square identical in size to the base (Figure 6.4). This size difference is the major modification on the half-cuboctahedron design. The plan view in Figure shows the base cable square with the top square above it, orientated anticlockwise at 45° to the base (Figure 6.5). Stability of this configuration is provided by diagonal cables extending from the bottom ends of struts, up to the top end of adjacent struts. These diagonal cables connect the base and top cables and ensures integrity of the form (Figure 6.6).

The initial design does not consist of any explicitly defined pre-stressed members, and it is necessary to note that the geometry described is not necessarily in equilibrium. It is rather an initial topology and the equilibrium form is still to be established. Member lengths are left unspecified at this point, since the purpose of this section is to describe the form rather than to establish the equilibrium geometry. Static equilibrium forms will be dealt with in detail in subsequent sections.

This particular four strut tensegrity is designed as a basic modular unit from which assemblies can be built. Setting the base and top level cable structure in the same square configuration allows modules to be stacked while

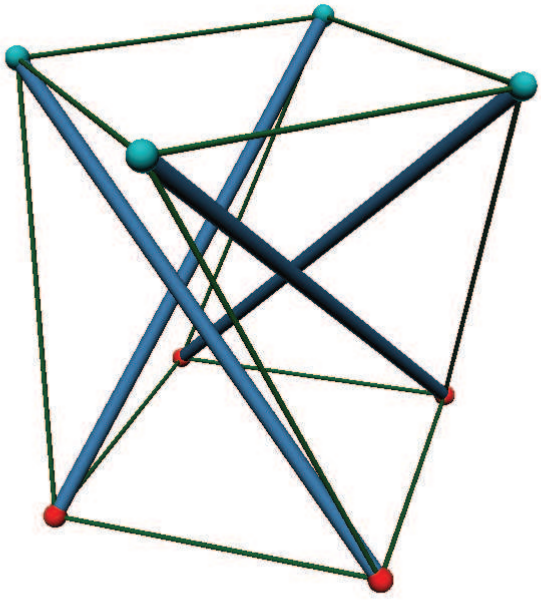


Figure 6.3: Four-strut module showing arrangement of struts

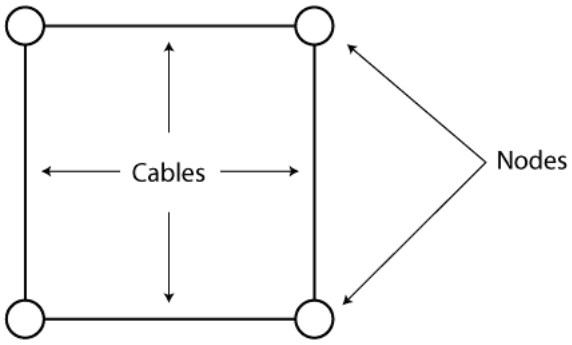


Figure 6.4: Four strut module base, showing nodes and cable arrangement

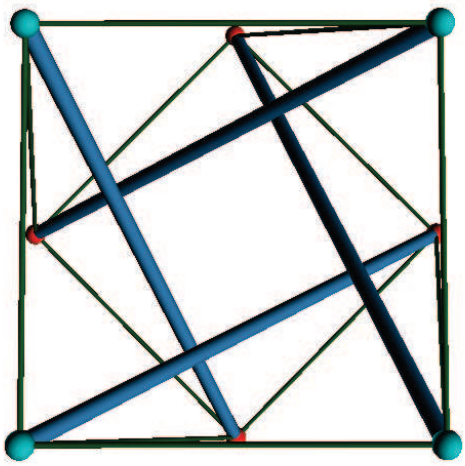


Figure 6.5: Plan view of four-strut module showing top orientation relative to base

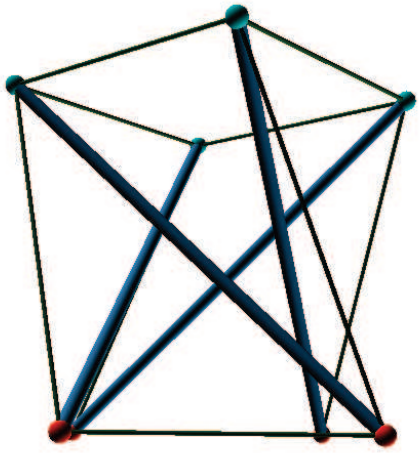


Figure 6.6: Elevation view of four-strut module showing diagonal cables

staying within tensegrity class one parameters. For instance, if two four-strut modules are stacked, their struts can be connected via cables extending from the top ends of struts int the lower module to the bottom end of struts in the upper module via cables. These cables will form a ring and allow the base and top cables to be removed from the top and bottom modules respectively. This allows the two modules to be connected without directly connecting the ends of struts, thus keeping the system class one. This modular concept can be extended further to create towers, or attached laterally to create grids (Figure 6.7).

The four strut module concept can be extended to forms with more struts. Using the same connectivity between struts and cables, the concept can be extended to produce other configurations, described by Motro [9] as "cylindrical forms". The more struts used, the more circular the module becomes, ultimately resulting in shapes with circular cross sections. Intuitively, increasing the number of struts would result in increasing the enclosed volume, and consequently arranging the cables towards the edges. Further extension and adaptation of this whole concept may give rise to cylindrical structures

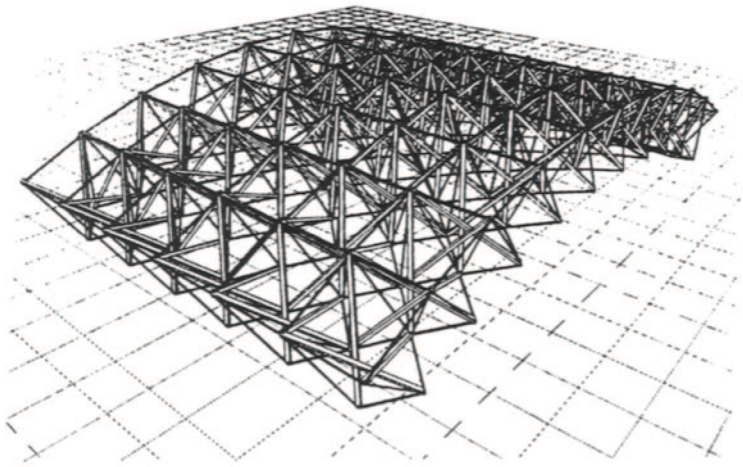


Figure 6.7: Double layer grid assembled from four-strut modules [9]

with double-layer strut walls in order to improve lateral stiffness.

The basic four strut module design, described above, is used as a second check on the validity of Tensegrity Designer and its numerical formulation.

The model was defined in Tensegrity Designer and loaded in a similar manner to the experimental testing scheme employed. The module was also modelled in ABAQUS under zero pre-stress configuration (See Input file in Appendix B). An initial displacement was applied to the top nodes to induce tension in the members (to remove finite mechanisms, see Section 6.1), thereby stabilising the system so that loads could be applied. Equal loads were incrementally applied to the top nodes, compressing the structure. The results of both models are shown in Figure 6.8.

As it can be seen, the shape of the graphs for the model versus the experimental data are similar. The results are in the same range and the curvature of the graphs are positive and increasing, showing an increase in stiffness as the load is increased. The first part of the graph of the ABAQUS model shows a zero slope. This results from the fact that finite mechanisms exist in the geometry under no initial loading. The result is therefore zero stiffness of the system, until a load is applied. Even a small load increment

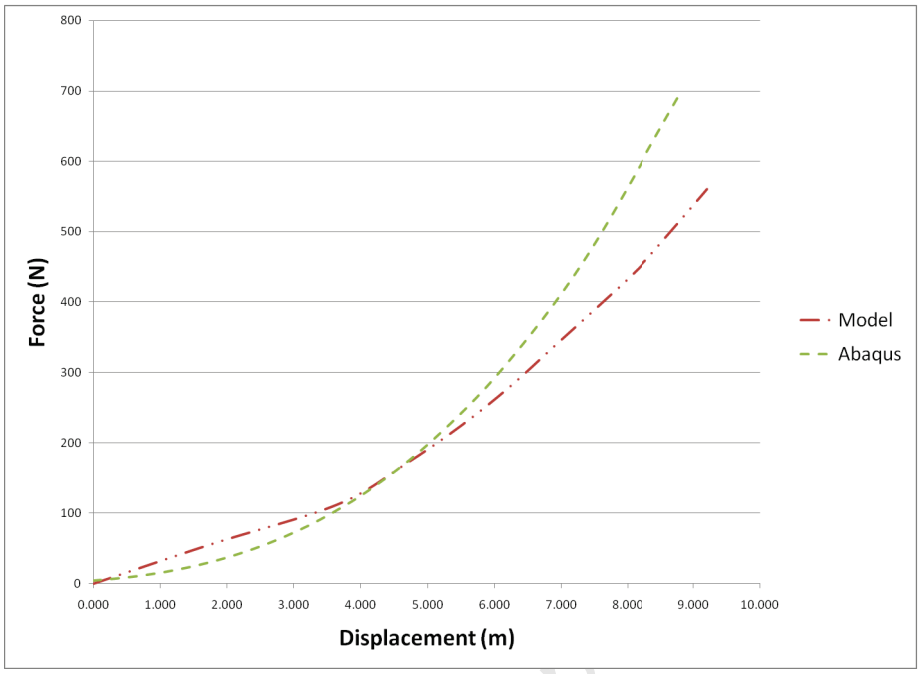


Figure 6.8: Force vs Deflection curve for four-strut module with no pre-stress

would result in initial deflections throughout the system. At which point infinitesimal mechanisms form and the system is under self-stress. As the load is increases, the stresses in the members increase, resulting in axial strain and an increased overall stress in the configuration. This causes an increase in stiffness which results in subsequent deflections being incrementally smaller as more loading is applied. This increase in stiffness under applied loading is one of the characteristics of tensegrity systems mentioned previously.

The difference between the results in the first part of the graphs can be accounted for by two factors. The initial applied displacement in the ABAQUS model and the different handling mechanisms for boundary conditions in either model. ABAQUS CAE requires additional boundary conditions at the base nodes to prevent rotation at the base of the structure. This would result in increased stresses on the elements and therefore a different load response profile.

6.3 Discussion

The results of the tensegrity software developed are in good agreement with the results obtained from ABAQUS. The load response in both models show the increased stiffness under increased applied loading. The general shape and quantities of the graphs correspond well and the differences in solutions are acceptably small.

The initial configurations modelled were not the equilibrium configurations. Deformations were applied in ABAQUS to essentially pre-stress the system, albeit only slightly. This resulted in an initial stiffness which allowed the application of loads, which under zero stiffness is not possible in the FEM environment. In Tensegrity Designer however, initial geometries with applied loading can be defined and the system can calculate the self-stressed equilibrium geometry. This form-finding process results in an equilibrium form which can be subjected to any loading conditions. This feature is not directly available in ABAQUS.

Although the results of the two models correspond well for the aforementioned conditions, it is important to note that this cannot be taken as conclusive evidence that the numerical formulations of Tensegrity Designer are correct. However, the results look promising and further models will be developed to determine the limitations of the program as well as its usefulness in the study of tensegrity structures.

Chapter 7

Model Fabrication, Construction and Testing

7.1 Four-Strut Module

7.1.1 Fabrication and Construction

The four-strut tensegrity module was the first model constructed to determine whether the envisaged fabrication system would be practical. This class one tensegrity module is assembled from twelve cables and four struts (See design specification in Section 6.2). The module consists of struts, cables and various connectors (Figure 7.1).

The struts were fabricated from aluminium circular hollow sections. At each end, two rows of 2.5mm diameter holes were drilled for the cable connections as shown in Figure 7.2.

Cables were cut from 1mm diameter stainless steel wire rope. This material consists of 7 strands of stainless steel cable, woven to form a 1mm diameter cable. The lengths of cable were prepared by crimping an aluminum connector at one end which would be used to secure the cable to one and of a strut (Figure 7.3). The other end would be secured via an

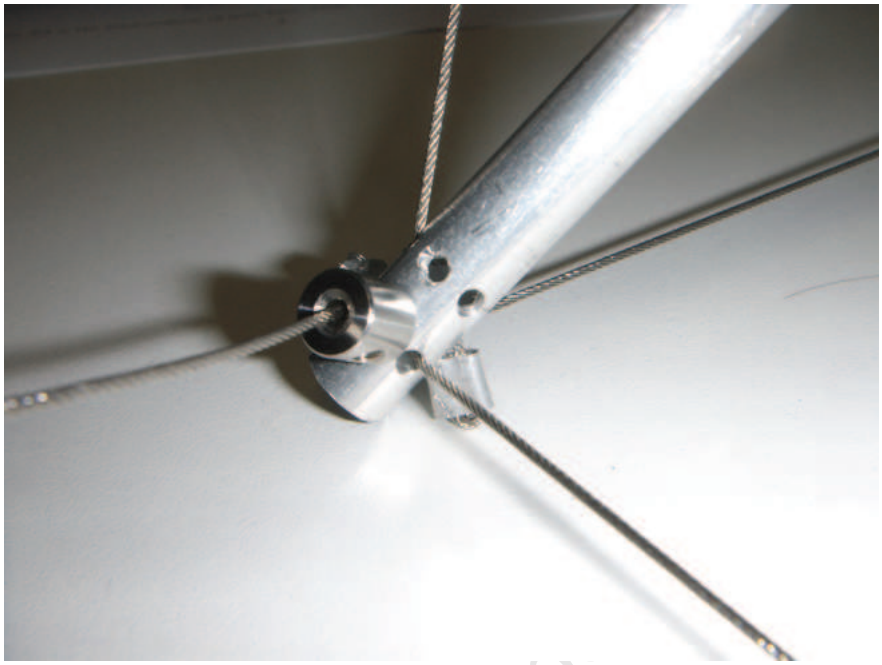


Figure 7.1: Base node showing cable and strut connections

adjustable connector, which allows pre-tensioning of the cable to occur and length adjustment after attachment.

Assembly of the module was as follows:

1. Cables and struts were cut to length. Fabrication lengths of struts were calculated based on design lengths, plus additional length was added for holes, since the node point is taken to be in the centre of the strut. Diagonal cables extend beyond this to the outside edge, where they are secured in place. This extra length of strut was taken into account and calculated based on hole size and hole edge distance. Cables were cut approximately 50% longer than design length. This was to allow for double folding of the cable at the crimped end and extra length at the tensioning end. Once secured in its final position, the extra unused cable length could be cut and removed.

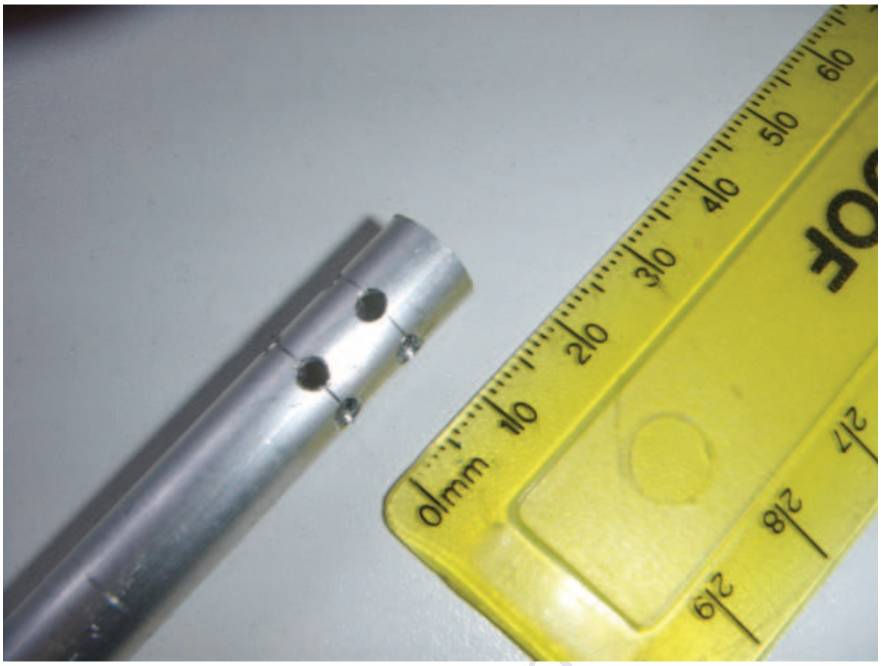


Figure 7.2: Strut end showing drilled holes for cable attachment

2. The base and top cables were attached. The base and top of the four-strut module are identical and was assembled accordingly. Four cables connect each of the lower ends of the struts together, similarly for the top cables. These cables were connected without taking into account design lengths since this would be adjusted later. The purpose was to secure the members in their proper position and then adjust the to the design length later, pre-tensioning where required.

3. The vertical cables were loosely attached. The four vertical cables were attached, each extending from the lower end of one strut, diagonally upwards to the upper end of the subsequent strut. These cables were attached slack, so that adjustment of the other cable lengths could be done easily.

4. Base and top cables were adjusted to their specified lengths. The base

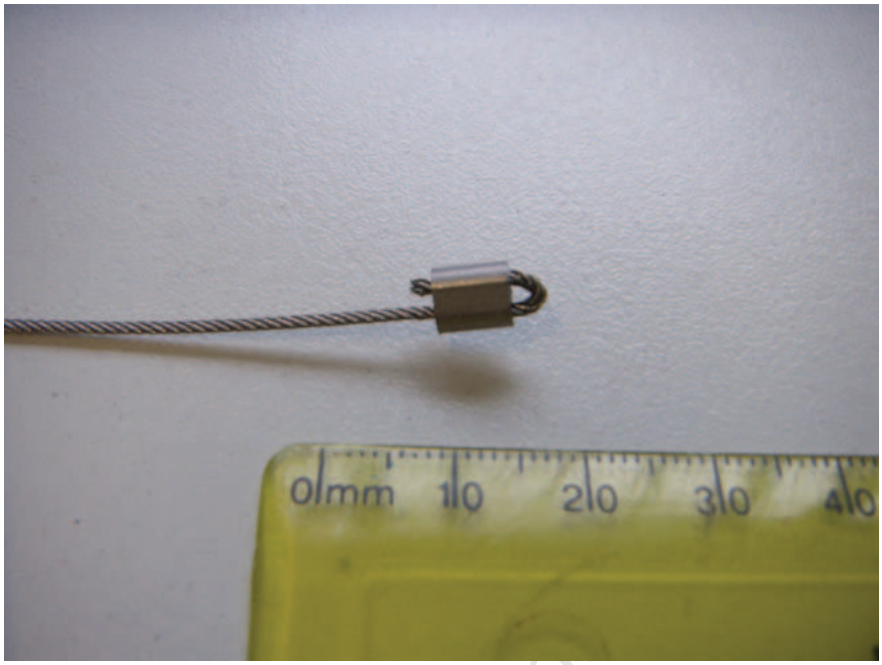


Figure 7.3: Crimped end of cable to connect to end of strut

and top cables were adjusted to the design lengths. At this point the system was still not stable.

5. Vertical cables adjusted to length and pre-tensioned. The vertical cables were adjusted to the design length. Adjustment of the final cable stabilised the system into a state of self-equilibrium. For this final connection, it was necessary for the system to be held in position before the cable could be attached. This required the application of force to keep the system in the configuration allowing the cable to be connected. Once the final cable was attached, the system moved to its stable equilibrium form (Figure 7.4).

This module was not designed to be under any specified pre-tension, but general fabrication errors resulted in the overall system being in a self-stressed stable form. Cables were found to be reasonably stiff, but none were under

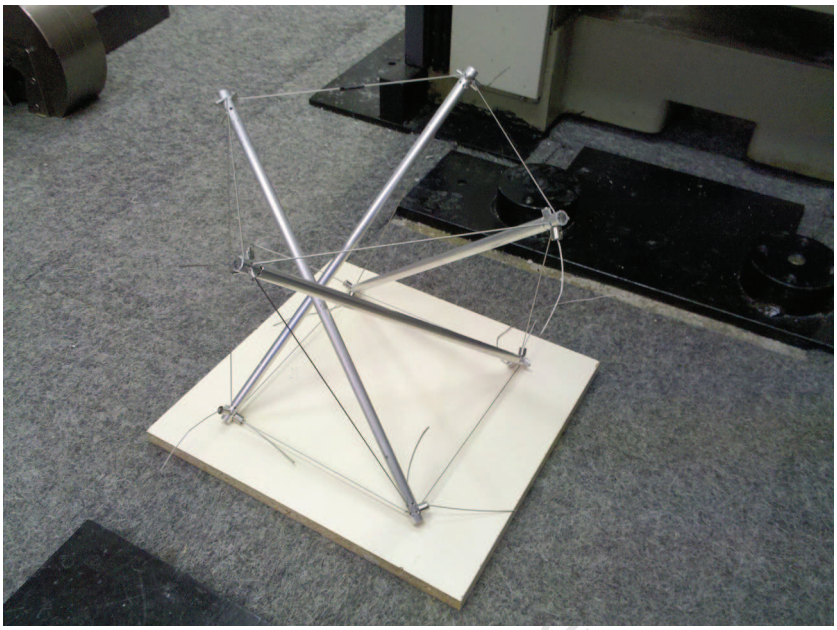


Figure 7.4: Completed four-strut module

any significant tension.

7.1.2 Testing

Looking only at the single four strut module, the loading conditions are intended to be applied only on the top nodes. On this basis, the boundary conditions for this module will now be defined. Since the tensegrity systems discussed in this dissertation are always pin-jointed, it is clear then that rotations about any axis cannot be fixed, i.e. only translational restraint is allowed on nodes. In this instance, the four nodes at the base layer will be restrained as follows. One of the base nodes will be fixed in all three translational directions. The other three nodes will each be restrained in the vertical direction, but will be free to move in both directions on the horizontal plane. These restraints sufficiently define the boundary conditions for this system.

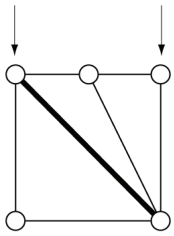


Figure 7.5: Compressive loading applied downwards on top nodes

As stated previously, loading will only be applied to the top nodes (No restriction is placed on loading the structure on a base node in an unrestrained direction, but this will not be required in our analysis). Loads may be applied in any direction and the structure is required to provide resistance to these loads in any direction or combination thereof. For our purposes we will apply multiple load cases and investigate the structural response. For experimental testing, only vertical loads will be applied and the vertical deflection of the top nodes measured (See Figure 7.5). Using Tensegrity designer, the same vertical loading will be simulated as well as additional load cases as described in the following chapter.

The four strut module was tested to compression loading in the Zwick testing apparatus (Figure 7.6). This was done to study the module's deflection response under applied loading on the top nodes. The module was place between two plates (Figure 7.7) to allow the force to be applied evenly on the four top nodes and to secure the model in the testing machine. Loading was done incrementally and set to end the test at a load of 600N. The results of this test can be found in Chapter 8.

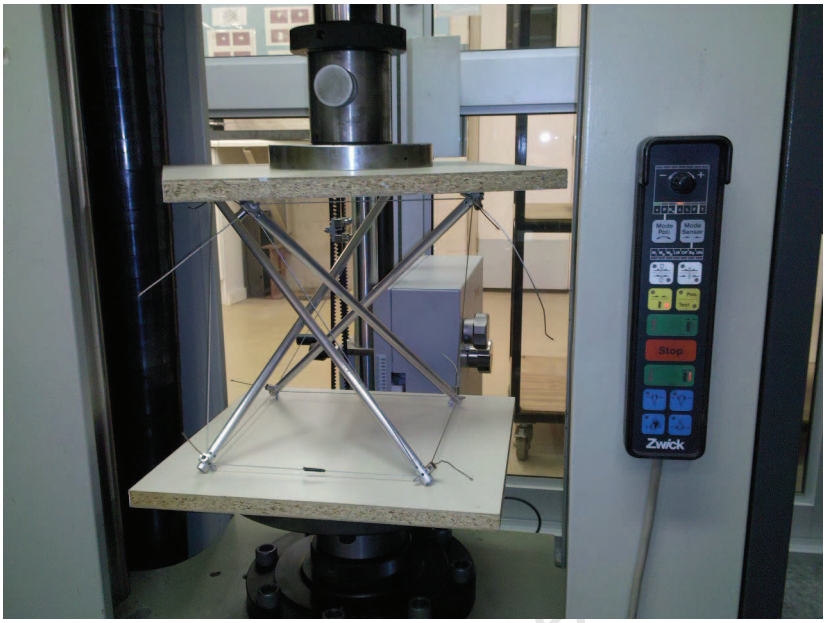


Figure 7.6: Four-strut module in Zwick Testing Apparatus

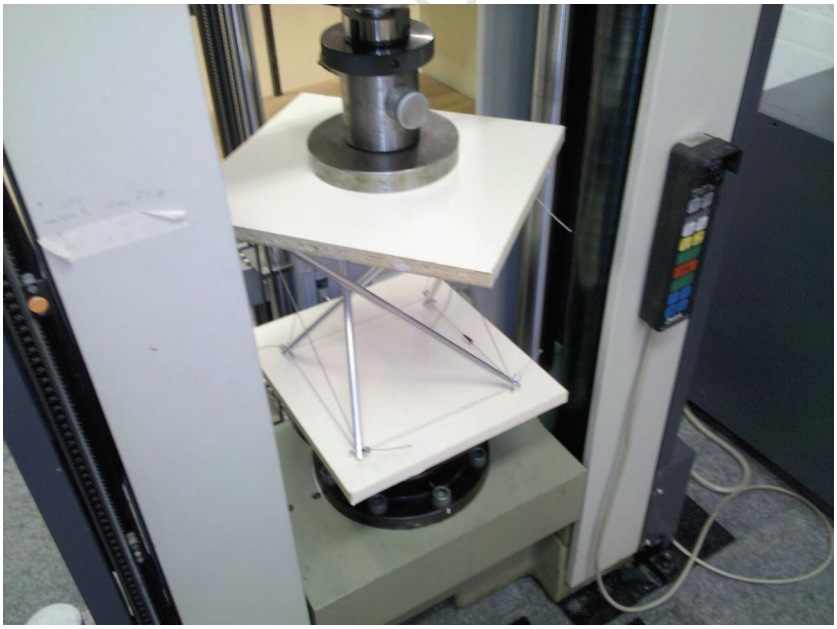


Figure 7.7: Module between two plates to allow even distribution of forces

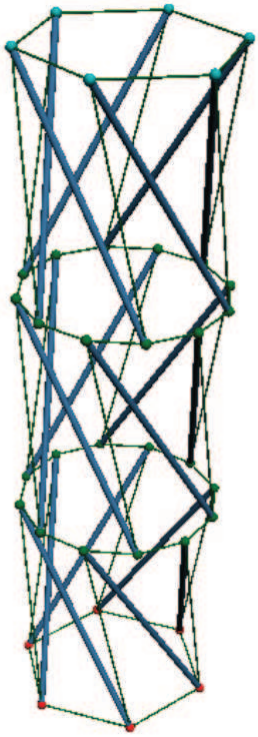


Figure 7.8: General tower arrangement assembled from stacked cylindrical modules

7.2 Tower

This design is a tensegrity tower built up from cylindrical modules mentioned previously (Figure 7.8). The modules are stacked, similar to the procedure proposed for the four strut modules. Two variables exist in this design, namely; the number of struts used per module and the number of modules in the column. The purpose of this design is to study tensegrity systems assembled from modules and their characteristics. Assembly of the column should provide insight into modular design, the problems and advantages associated with it. A model of this tower was designed and built in order

to investigate the aforementioned aspects. The complete model (See Figure 7.9) was then tested experimentally and the results recorded for comparison with the programmed design and its results.

Six struts were used per module (Figure 7.10), with three modules being stacked to create the tower. All the boundary conditions and load cases associated with the four strut module applied similarly to this system. Additionally, changes in stiffness characteristics will be investigated based on number of struts used per module, as well as number of layers of stacked modules. The behaviour of the structure do to the increased internal volume and height will also be studied.

This design has numerous possible applications. It can be used as a column member as part of a tensegrity structure and may be equally functional used in traditional structures. Using the variables as parameters mentioned previously, it is envisioned that the column can be designed to be as tall and as stiff as required, within geometric and material limits. Another application would be to use the system as a beam or girder. Pre-stress can be applied to create pre-tensioned beams capable of withstanding higher loads with less deflection. These two basic types of members; columns and beams, can be combined in construction systems which are comparable to conventional structures. It is however important not to restrict tensegrity systems to this type of design methodology since many advantages can be gained by designing structures on the basis of tension.

The column design can be easily scaled up to produce larger cylindrical structures such as towers or masts. Grid-like roof systems can be developed by spanning the cylindrical tubes parallel and perpendicular to each other and using the long spanning members as girders. These roofs may be given single or double curvature depending on the application, for example a tensegrity dome structure.

All of the above applications result from adaptations of the basic tower modular design. It is apparent that any application would have numerous re-

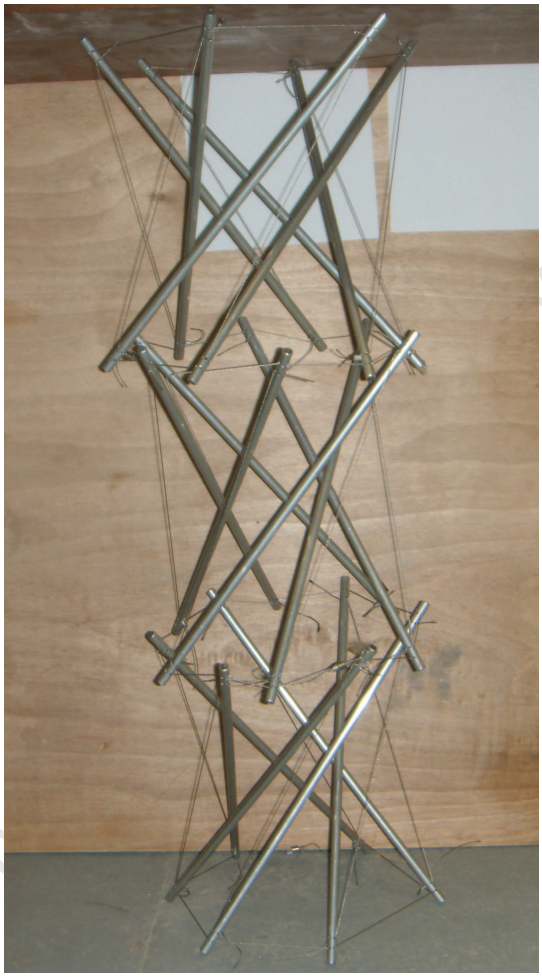


Figure 7.9: Completed assembly of tower model

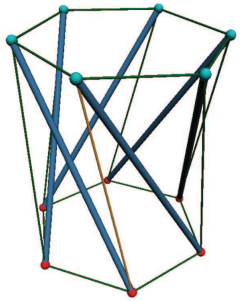


Figure 7.10: Six-strut base module used in tower assembly

quirements, including strength and stiffness, and practical application would therefore be greatly influenced by these parameters. It is therefore important to investigate the behaviour of these types of structures, given different parameters, forms and external actions. This will be the primary focus in the following chapters.

7.2.1 Fabrication

The second model to be fabricated and constructed is the three layer, six strut modular tower (See Section 7.2 for design specifications). The tower was assembled similarly to the four strut module, taking into account the lessons learnt from the four strut module. It was seen that the crimped end of cables (Figure 7.3) was a weak-point of the design. It was most likely to fail here by detachment of the aluminium crimp connector. It was therefore decided to change this end connection to a double loop around the connector to eliminate any chance of slip-out of the cable. Also, the edge distance of the strut holes were increased to 15 mm to reduce the chance of loading on the connection and ensure that all applied loads are taken by the strut end. The design comprises:

- 18 x 430 mm Aluminium circular hollow sections, 10 mm diameter, 1.2 mm wall thickness

- Various lengths of cables
- Crimping Connectors

The struts were cut to length and fabricated similarly to those of the four-strut module. The edges were smoothed out to reduce friction between the plate during testing, and consequently not fix movement of the strut ends in the horizontal plane. Cables were cut and their ends dipped in hot wax to form a protective coating preventing fray of the strands.

7.2.2 Assembly

The model was assembled from the bottom module up. A single cable was used to connect the ends of modules, thereby shorting the cable-preparation time, and allowing easier cable length adjustment and pre-stressing. The horizontal cables at the top and bottom were attached loosely and then the vertical cables were connected at their design length. This procedure was repeated for the middle and top modules. Once all the cables had been connected, stabilisation of the structure could commence. This was performed by tensioning the horizontal cables forming the base, top and inter-module connections between strut ends.

7.2.3 Testing

A plate was placed over the top of the tower as a platform for weights to be loaded upon. It also ensures that the applied load is distributed evenly onto the top ends of struts in the top layer. Loading was done manually by placing weights one at a time onto the plate. After each load was added, the deflection was measured and recorded. The results can be seen in Section 8.1.2.

Chapter 8

Results

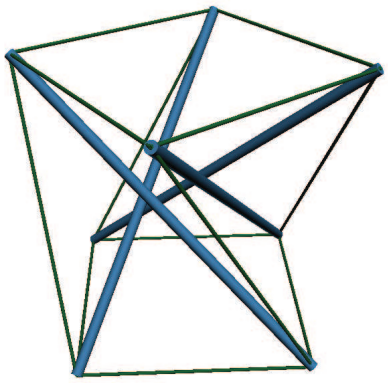
8.1 Experimental

Experimental tests were performed on the models whose fabrication was described in the previous chapter. Compression tests were performed on these models and the results are now presented.

8.1.1 Four-strut module

The following are the results of the compression test performed on the four-strut module as described in Section 7.1.2. The results are plotted together with the predicted values from the programmed model.

The results are generally in agreement with some variation in the lower force range. The shape of the two result graphs are similar in curvature, showing agreement between the displacement response. The increasing gradient of the curves shows that the stiffness is increasing under increased load application. This phenomenon results from the distribution of forces through the continuous cable system, and the resulting tension in the cables cause



(a) 3-Dimensional graphic

Tension Member Section	1 mm ϕ Stainless Steel Cable
Vertical Cable Length	284 mm
Horizontal Cable Length	212 mm
Strut Section	10 mm ϕ Aluminum 1.3mm Thick CHS
Strut Length	380 mm
Model Height	150 mm

(b) Member and overall dimensions

Figure 8.1: Four-strut model structural specifications

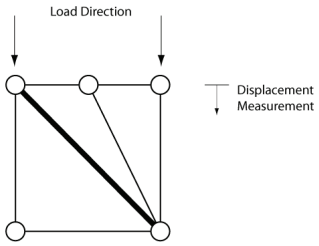


Figure 8.2: Four-strut model elevation showing load application and displacement measurement directions

Total Vertical Force (N)	Displacement (mm)
0	0.0
56	1.8
112	3.6
168	4.6
224	5.5
280	6.2
336	6.9
392	7.5
448	8.2
504	8.7
560	9.2

Table 8.1: Four-strut model compression loading results

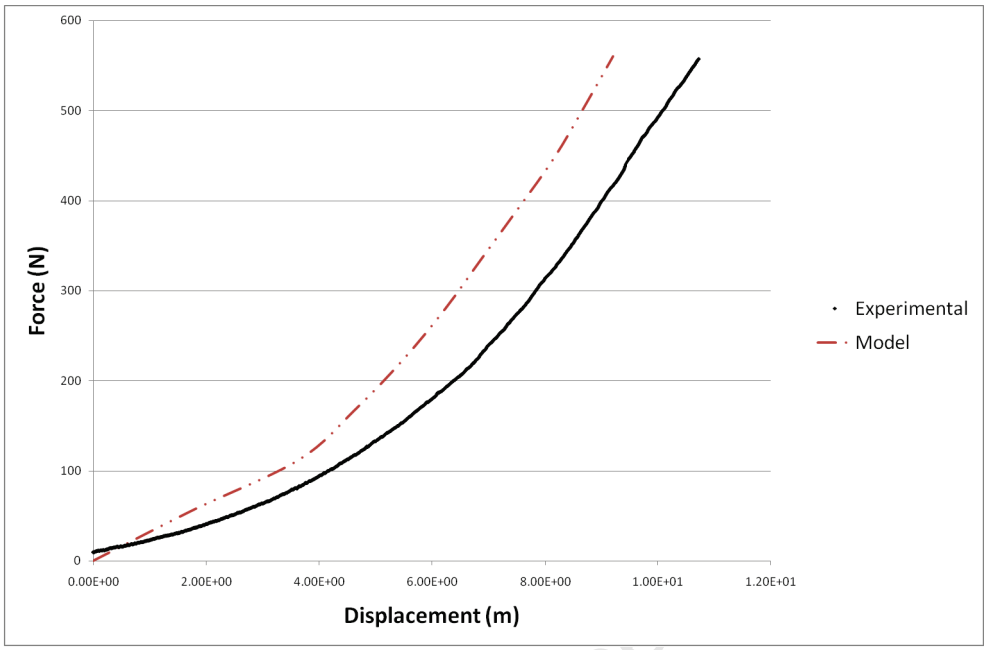


Figure 8.3: Four-strut model compression loading results chart

an overall increase in stiffness in the direction of loading. Although the two result sets show similar trends and displacement behaviour, there are differences such as the sudden change in curvature of the programmed model at approximately 100N. The change can be explained by noting the manner in which the cables become resistive to force. As explained in Section 6.1, cables which are not pretensioned have zero initial lateral stiffness. However after initial application of loading, the resulting tension in the cables cause lateral resistance. Therefore the change in gradient of the programmed model graph is the result of certain cables gaining stiffness properties due to tension forces that have been distributed through their length. The smooth curvature of the experimental results are due to pre-tension forces in the physical model. Since the cables are not slack and have pre-existing forces, there is no lag time during which the cables build up tension as in the numerical model. Other factors which may cause deviation in the numerical (programmed) results

from the experimental data are:

- Fabrication length errors
- Cable slip at connections
- Connection offsets from actual modelled location
- Unknown pre-tension in members
- Initial deflections due to fabrication error

8.1.2 Tower

Below the results of the compression test on the constructed tower, as described in Section 7.2.3. The results are plotted against predicted values from the programmed model.

The experimental results are generally in the region of the programmed model. The curvature of the experimental results in the lower force range exhibits increasing stiffness under the application of loading, whereas the general trend displays a decrease thereof. Pre-tension forces in the members of the constructed model contribute significantly to this deviation. The actual pre-stress of the members cannot be accurately quantified in practice. Therefore various models were developed and programmed in Tensegrity Designer, with different pre-stresses in the members. This was achieved my applying displacements to the vertical cables which, after form-finding, gives the overall equilibrium configuration. The stresses of the vertical cables are automatically distributed throughout the members in the system in order to acquire the self-stressed equilibrium form. The results are shown in Figure 8.7.

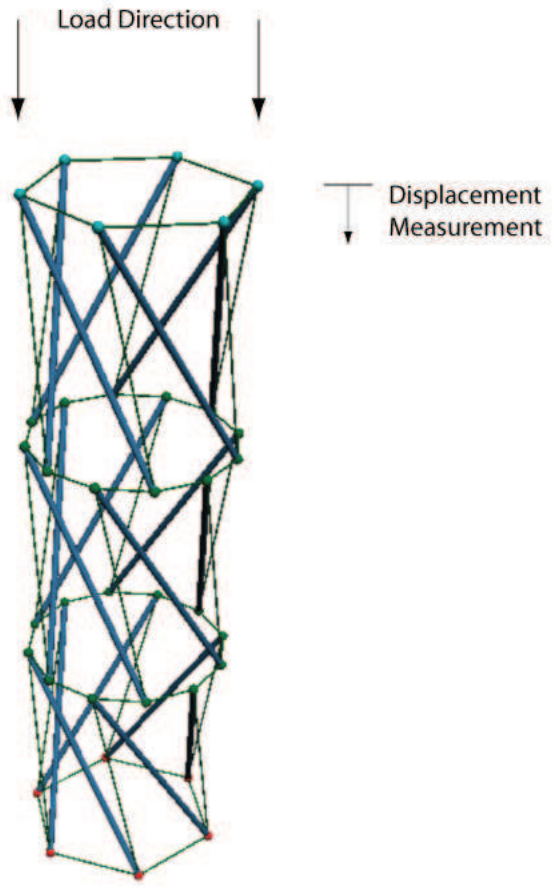


Figure 8.4: Tower model 3-Dimensional graphic showing load application and displacement measurement directions

Tension Member Section	1mmϕ Stainless Steel Cable
Vertical Cable Length	349 mm
Base Cable Length	150 mm
Inter-layer Cable Length	78 mm
Strut Section	10mmϕ Aluminum 1.3mm Thick CHS
Strut Length	400 mm
Model Height	1020 mm

Figure 8.5: Tower model member and overall dimensions

Total Vertical Force (N)	Displacement (mm)
22	0.0
44	3.0
68	5.0
90	7.0
112	12.0
135	14.0
157	15.5
180	19.0
203	21.0
225	23.0
305	25.5
385	30.0
465	35.0
545	47.0
625	48.0
705	51.5

Table 8.2: Tower compression loading experimental results

Total Vertical Force (N)	Displacement (mm)
30	0.1
60	1.0
90	2.9
120	4.7
150	7.5
180	9.6
210	12.3
240	15.0
270	18.7
300	22.3
330	26.2
360	30.5
390	33.8
420	38.6
450	41.3
480	45.6
510	50.5
540	55.5
570	60.4
600	65.9

Table 8.3: Tower compression loading results from computational model

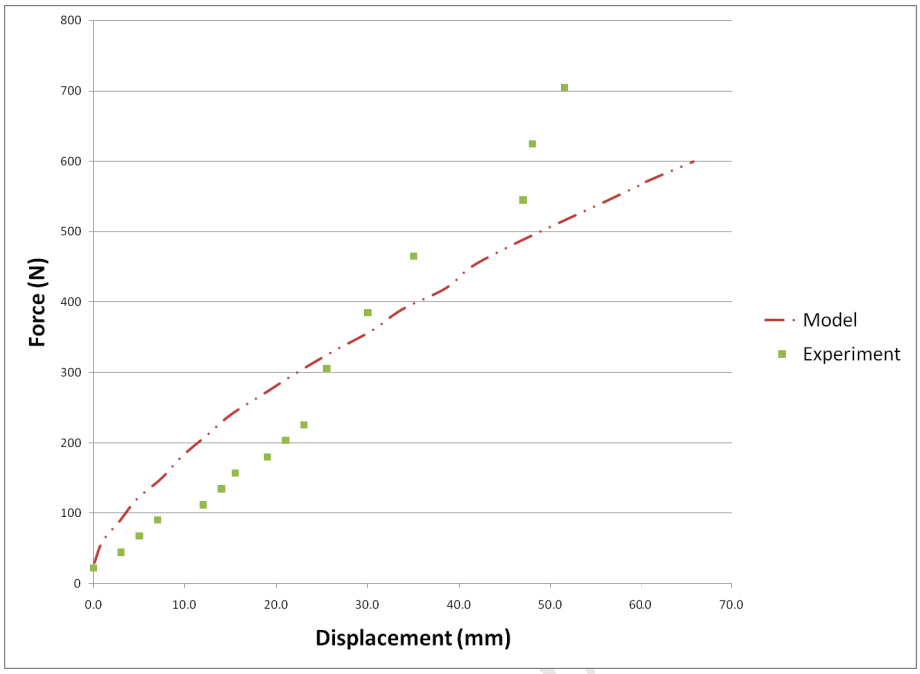


Figure 8.6: Tower compression loading results chart

8.2 Form-Finding

An example of the numerical (programmed) model's form-finding procedure is shown in Figure 8.8.

8.3 Load Cases

The following load cases will be investigated via programming and analysis using Tensegrity Designer:

- Vertical tensile loads applied to the top nodes to investigate the tensile characteristics of the module (Figure 8.9a).
- Lateral loading in one direction only. Applied in order to test the systems resistance in bending and its lateral stiffness properties (Figure 8.9b).

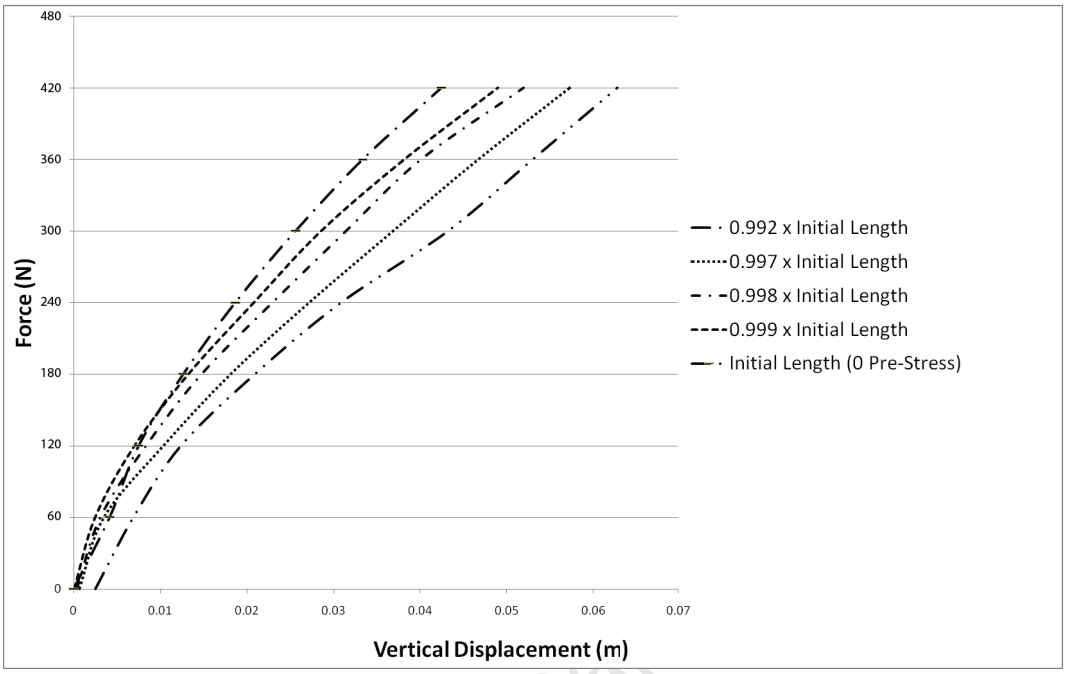


Figure 8.7: Tower model under various self stress levels induced by reducing initial length of the vertical cables

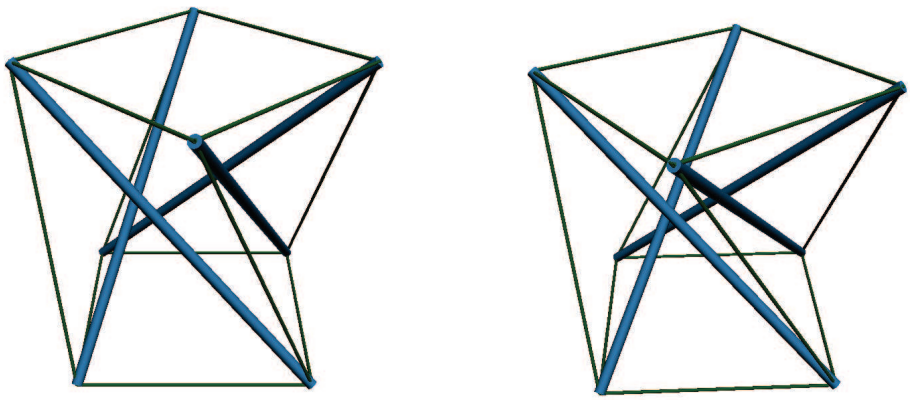


Figure 8.8: Initial (left) and equilibrium (right) configurations of four strut module

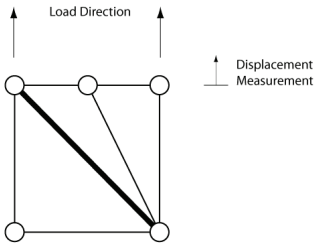
- Various combinations of loading on top nodes towards the centre of the module, including only opposite nodes as well as all nodes ("squashing", Figure 8.9c).

The results that are required from the above load cases comprise the final equilibrium form under the various load conditions and the trace of the displacement of the structure. This includes nodal deflections, member extensions or compressions, and member internal stresses. In order to analyse load response and investigate stiffness characteristics, the deflection response under increasing load application is required. Thus for certain load cases, where deemed appropriate, loads will be applied incrementally and the deflection response of the nodes recorded after each increment. This data can be analysed to investigate behaviour such as increased stiffness of the system under increased applied loading. Member forces are required to determine load distribution in the structure and to determine material stresses and failure. This information is critical in the design of tensegrity structures.

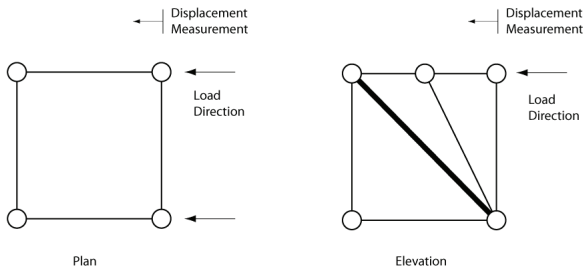
The remaining load cases presented for the four-strut module (Section 6.2) were modelled in Tensegrity Designer and the results are presented below. All displacements were taken as their absolute values to provide a reference point for comparisons between the various load cases and their stiffness responses. For example, between compressive and tensile loading conditions.

8.3.1 Vertical Tensile Loading

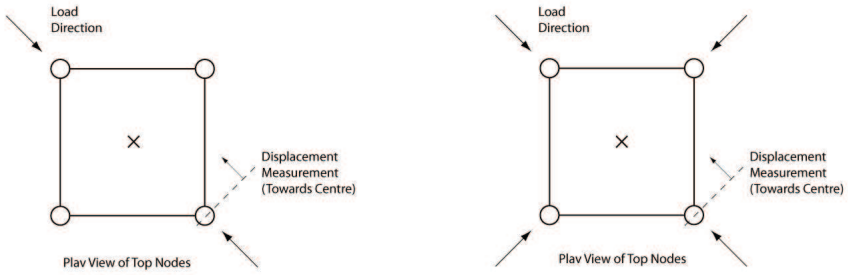
Figure 8.10 shows the results of the applied vertical tensile loading on the four-strut module. Each of the top nodes were loaded equally and incrementally increased to a total of 100N. The results obtained show a non-linear relationship between the force applied and the resulting displacement. On initial application of loading, there is minimal displacement of the system up to 15 N, at which point initial displacement occurs. This displacement is due to loss of stiffness of the system, causing the rate of displacement to



(a) Tensile loading applied upwards on top nodes



(b) Lateral loading in one axis only



(c) Squash loading in one and two axes

Figure 8.9: Load cases for four-strut module applied to top nodes.

Total Vertical Force (N)	Displacement (m)
0	0.000
10	0.000
20	0.001
30	0.001
40	0.002
50	0.004
60	0.005
70	0.006
80	0.006
90	0.006
100	0.007

Table 8.4: Results from computational model for tensile loading of four-strut module applied to top nodes

be increased. As the load is further increased, the module deforms and vertical displacement of the top nodes occur up to approximately 90N where stiffening occurs and displacement is minimal upon further increase of the load.

The overall force-displacement response for this load case is an extended "S"-shaped graph. Initially there is minimal displacement of the nodes as the force is applied, due to the resistance provided by the struts. As the force is increased, the struts displace laterally and transmit the forces through the top level horizontal cables. At this point displacement occurs as the tension in the cables increases. This increase is gradual up to the point at which the the forces in the cables result in tensile resistance, stiffening the module. Cable tension therefore dominates the resistance contribution at this applied force and further increase of the force results in minimal displacement due to the high tensile resistance of the cables.

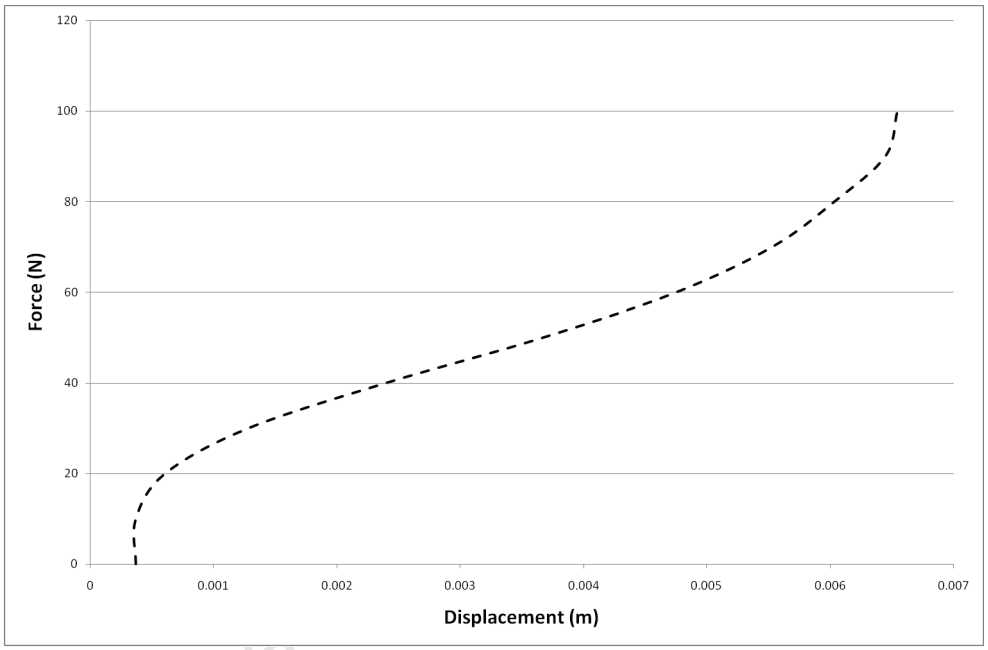


Figure 8.10: Force-displacement response for tensile loading of four-strut module applied to top nodes.

8.3.2 Lateral Loading

The four-strut module was loaded on two top nodes in the direction of the centre of the module. Equal forces were applied to the two nodes and increased incrementally. The displacement response under the applied loading is plotted in Figure 8.11. The response show a decrease in stiffness as the load is increased, but this decrease is negligible and the response can be approximated as linear. This linear response indicates that under this type of loading, displacement will increase at the same rate at which loading is increased, and that the stiffening effect would be negligible. This behaviour is due to the direction of loading and the resistances of members in this direction. The loading pattern tries to compress certain cables and the force direction is perpendicular to others. This pattern results in a low resistance contribution from the cables, causing the struts to take almost all of the loading. The displacement profile shown in Figure 8.11 is therefore due to strut movement with essentially no resistance contribution from the tensile members.

8.3.3 Squashing in one axis

In this load case, two opposite nodes are loaded equally in a direction towards the centre of the module. Once the initial resistance of the module is overcome and redistribution of forces takes place, the tensile members become stressed and the induced tension in these members result in increased resistance to deformation. This only takes place after deformation of the system to such an extent as to tension all the members. The resulting behaviour and displacement profile (Figure 8.12) is similar to the tension load case (Figure 8.10) exhibiting a "S"-curve. This "S"-curve behaviour is typical of tensegrity systems.

Total Vertical Force (N)	Displacement (m)
0	0.000
10	0.000
20	0.000
30	0.003
40	0.006
50	0.010
60	0.013
70	0.017
80	0.021
90	0.025
100	0.029

Table 8.5: Results from computational model for lateral loading of four-strut module applied to two adjacent top nodes.

Total Vertical Force (N)	Displacement (m)
0	-0.000431
14	-0.000402
28	0.000327
42	0.001656
57	0.003807
71	0.006321
85	0.010178
99	0.011445
113	0.012595
127	0.013151
141	0.013283

Table 8.6: Results from computational model for squash loading on four-strut module applied to two opposite top nodes.

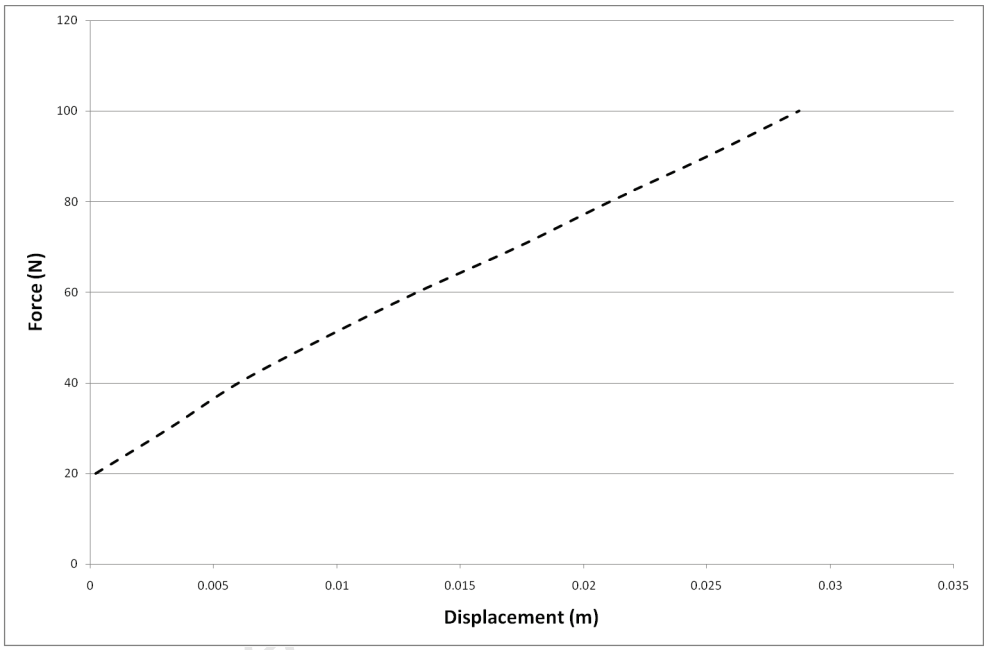


Figure 8.11: Force-displacement response for lateral loading on four-strut module applied to two adjacent top nodes.

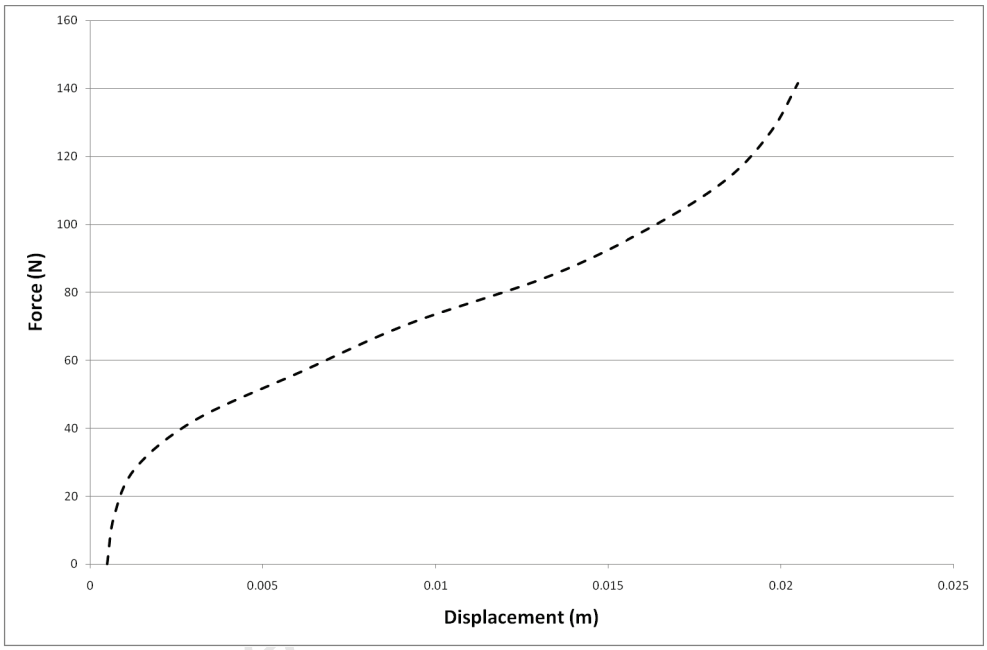


Figure 8.12: Force-displacement response for squash loading on four-strut module applied to two opposite top nodes.

Total Vertical Force (N)	Displacement (m)
0	0.0000
14	0.0001
28	0.0002
42	0.0004
57	0.0004
71	0.0006
85	0.0006
99	0.0007
113	0.0009
127	0.0009
141	0.0010

Table 8.7: Results from computational model for squash loading on four-strut module applied on all four top nodes.

8.3.4 Squashing in both axes

When all four top nodes are loaded in a direction towards the centre of the module, the result is approximately no displacement of the nodes in the direction of loading (Figure 8.13). The forces applied in each axis tension the cables and provide resistance to loading in the perpendicular axis. The forces essentially "cancel out," thereby allowing negligible displacement. Cylindrical tensegrities, like the four-strut module, consequently offer high resistance under this type of symmetrical loading conditions.

8.4 Numerical Modelling with Tensegrity Designer

Numerical models were developed using generic forms which could be parameterised in terms of strut count and module arrangement. Both cylindrical

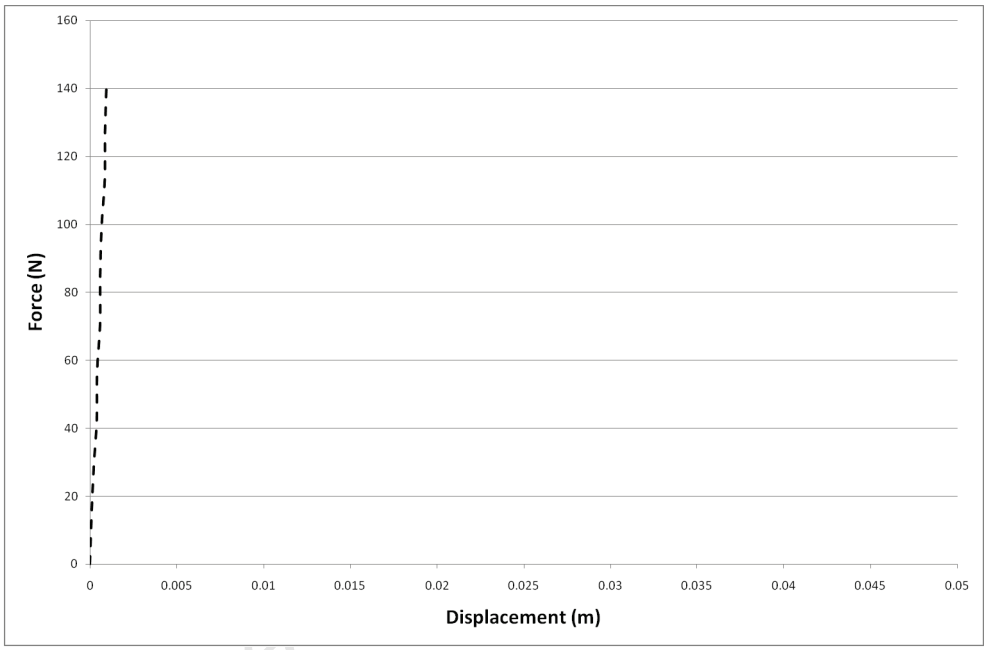


Figure 8.13: Force-displacement response for squash loading on four-strut module applied on all four top nodes.

modules and towers were investigated using this method. The parameterised models were analysed and the results thereof are presented below.

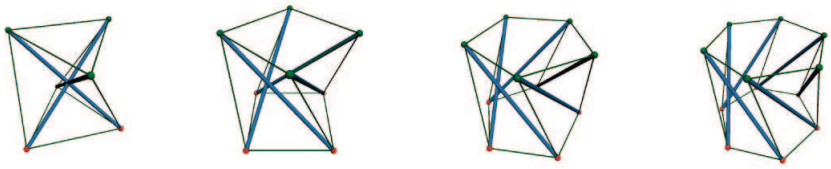
8.4.1 Compression Test on Modules

Diamond tensegrity modules were modelled for configurations composed of different strut counts and configurations. This was done to determine the load response and stiffness behavior that the modules exhibit depending on the number of struts they are composed of. The height and total load were kept constant. For each module, the load was distributed equally on the top nodes and applied in an increment fashion until the total load was reached. The results for displacement and stiffness are shown in Figure 8.14.

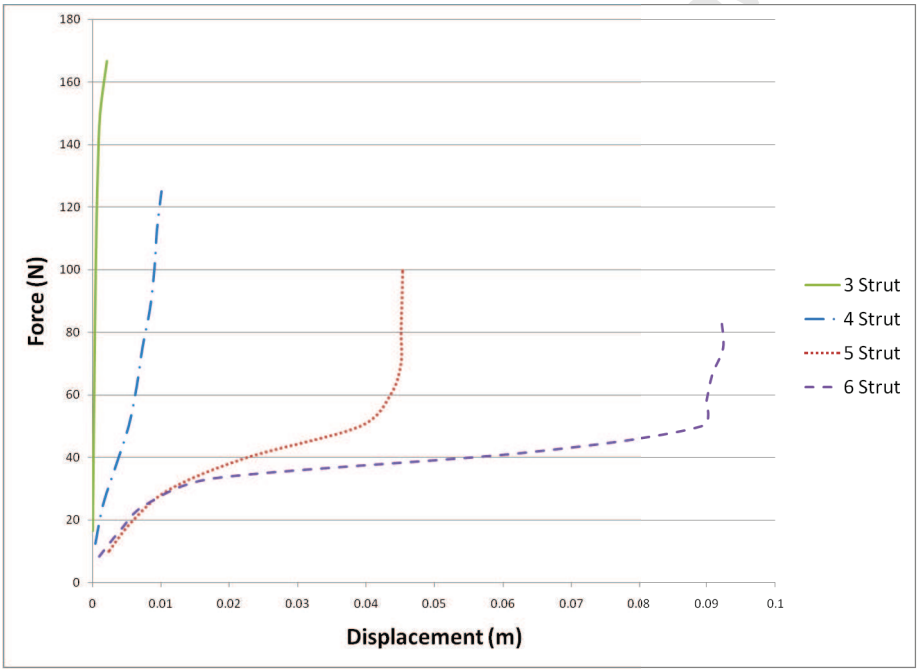
Comparing the different configurations, it is clear that as the total number of struts per module increases, the deflection under loading increases for the same load magnitudes. The force-displacement response type is similar for the different modules, with stiffness in the loaded direction generally increasing as the load is increased. The similarities in load response can be identified by the "S"-curve, as seen in Figure 8.14 for the five and six-strut modules respectively. Generally it is noted that increasing the strut count in the module consequently decreases its load resiting capability. Displacements are significantly larger for modules with more struts and stiffness is comparatively higher for modules with fewer struts.

8.4.2 Compression Test on Tower

Towers with constant height but varying layer counts were modelled to determine the effect of additional layers on load carrying behaviour. The total load applied was kept constant and distributed on the top nodes. Load was



(a) Module geometry for three, four, five and six struts, respectively



(b) Compression test results chart

Figure 8.14: Compression test results for modules of increasing strut count

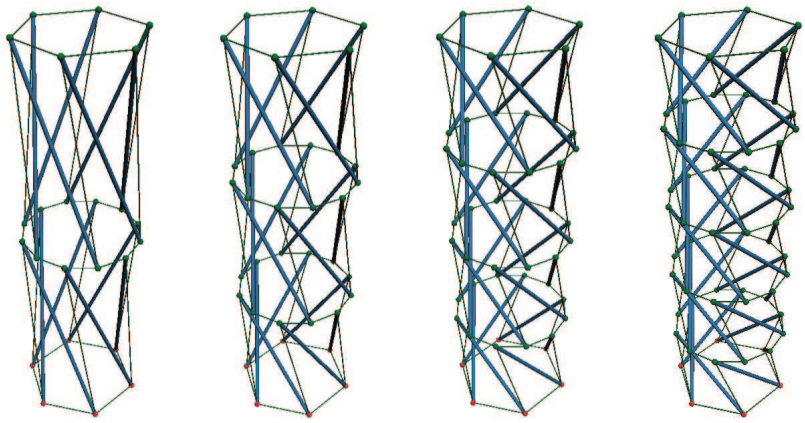


Figure 8.15: Tower geometry for two, three, four and five module layers, respectively

applied incrementally for two, three, four and five layer tensegrity towers respectively. The results are shown in Figures 8.16 and 8.17.

The load response for the models were in the same region up to a critical load, at which point the two and three layer towers exhibited increased stiffness compared to the four and five layer models. It has to be noted that in general the load response is fairly similar when increasing number of layers, yet this small variation is advantageous as it allows the use of shorter compression members, thereby reducing the likelihood of local buckling failure.

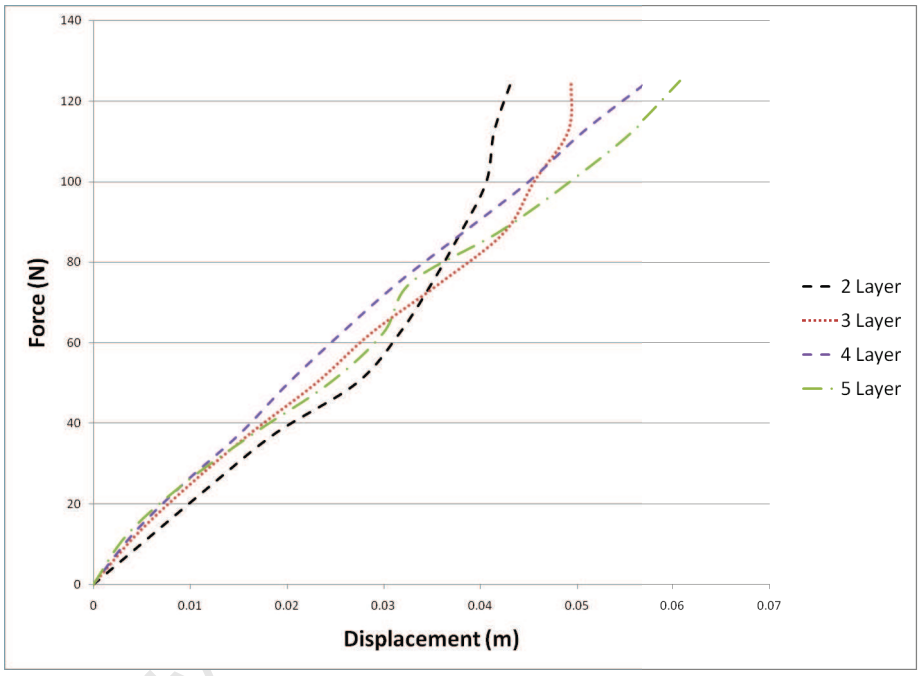


Figure 8.16: Force versus displacement chart for multi-layer towers

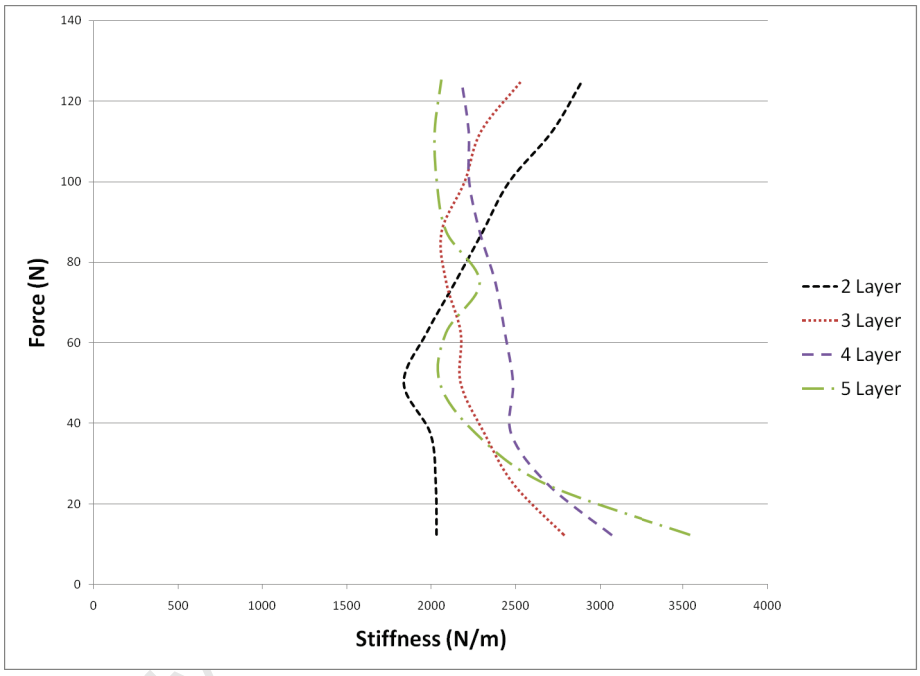


Figure 8.17: Force versus stiffness chartfor multi-layer towers

Chapter 9

Conclusions and Recommendations

This dissertation primarily studies the form-finding and analysis of tensegrity systems. Theory has been presented and adapted to tensegrity systems to design tensegrity structures and study their behaviour. Here follows the most important observations, conclusions and recommendations for future work.

Tensegrity structures in general are particularly interesting with regards to efficiency, mass to strength and deployable construction. New tensegrity structural forms, their design and implementation would give rise to a new generation of structures with a focus on lightness, aesthetics and structural stability.

The development of a reliable, robust and computationally efficient design system for both modular and irregular tensegrity structures is both appealing and necessary if tensegrity structures are to be practically implemented by design engineers and architects. At the present time there is no universal method of form-finding and analysis for tensegrity structures, nor any fabrication and construction guidelines. The introduction of these features in the field of tensegrity would open the doors to the construction of increasingly innovative and revolutionary structures. Existing technology is predomi-

nantly developed for traditional compression structures, and those that do offer tension structure design features are rarely adaptable to tensegrity systems. An opportunity therefore exists for the development of a generalised, form-independent tensegrity design system. Tensegrity is a relatively new concept in industry and thus the initial costs involved in its implementation are high.

Chapter 4 formulated the dynamic relaxation method for the form-finding and design of tension structures and its application to tensegrity systems. This technique is simple to implement and computationally efficient. The quasi-dynamic approach can be reliably used to analyse tensegrity systems. Chapter 5 presents an implementation of the dynamic relaxation method in a Visual C# Application which I specially developed for this thesis. The software allows form-finding and analysis of tensegrity systems. It is used to show the relationships between load and stiffness of tensegrity models, as well as the correlation between number of struts and compression strength for diamond tensegrities. These numerical models showed realistic behaviour when compared to experimental results.

9.1 Conclusions

The primary reasons for studying the behaviour of tensegrity structures is to make progress on the road to increased usage of this technology in industrial applications. A thorough understanding of tensegrity systems, their characteristics and behaviour will result in predictability in modelling and increased potential for usage. This will in turn accelerate development of design systems, fabrication techniques and equipment to reduce the complications associated with the construction of these types of systems.

This dissertation successfully modelled various tensegrity models and studied their behaviour under various configurations, internal and external actions. Application of the dynamic relaxation method with kinematic damp-

ing was applied to tensegrity geometries to solve the form-finding problem as well as determine static equilibrium configurations for applied loading and boundary conditions. This technique was used to develop a software application to apply the formulation and obtain results from it. Models were defined to determine load response in different directions and orientations of the systems. Geometries were parameterised and modelled to determine the effect of modification of these parameters on overall system behaviour. Two designs were fabricated, constructed and experiments conducted to determine some of the complications and also the advantages associated with construction of tensegrity systems. Experimental data was compared to their numerical model results to determine the level of agreement and reasons for divergence.

It was found that tensegrity systems exhibit increased stiffening as the applied load is increased, as predicted by the literature reviewed. However in some instances this behaviour was not found in the numerical modelling results. This was due to the amount of pre-stress in members in the equilibrium configuration as well as the boundary conditions in the model. It was shown that increasing the amount of struts in diamond tensegrity modules results in a decrease in relative stiffness between members of the same general form.

The dynamic relaxation technique proved to be a reliable system for determining equilibrium states and forms. The advantage to using this method is that assembly of the global stiffness matrix was not required for analysis. This decreased computational effort and simplified programming the design software. Tuning of the λ parameter allowed convergence of the solution to be optimised for different models.

This dissertation successfully showed that irregular tensegrity structures can be designed successfully using the dynamic relaxation method. Form-finding and analysis are included in this design process, and the results thereof can be used in opening the way for further contributions to the design of tensegrity systems.

9.2 Recommendations

Tensegrity structures is a rapidly growing field with many challenges to solve and much research taking place. It has great potential for application for architecture and engineering.

Improvements can be made to the dynamic relaxation algorithm proposed here by automating the optimisation of the mass factor. Ensuring optimum convergence would decrease model development times and the time it takes per analysis. It would also result in more accurate modelling and decrease instances of system divergence due to incorrectly estimating mass factors. This improvement would also allow more complex configurations and loading conditions to be modelled without sacrificing computational efficiency.

Extension of the tensegrity concept through adaptive control systems and automated configuration changes have been studied by others. Implementation of these features into the tensegrity design software would open the door to increased usage and application of smart tensegrity structures.

This dissertation focused on the development of designs based on strict class one tensegrity criteria. Class two structures do however offer advantages in the areas of structural stiffness due to the inter-connectivity between compression elements. Further work may be carried out to determine the feasibility of using class two instead of class one for various types of structures. That said, each class in itself provides advantages over the other and it is up to the designer to determine which system to make use of, if not both.

Dynamic analysis and vibration control is an important area that needs to be studied carefully for tensegrity systems. Their lightweight nature and variable stiffness characteristics result in increased vulnerability to dynamic failure. Failure criteria is an important design aspect which requires further work. Establishment of failure modes and limit states for tensegrity structures is critical to the application of these systems.

The practical design and construction of tensegrity structures for commercial use is still a new and relatively untapped field. Much research effort

is still required to take the concept into industry. Research and experimentation into building useful tensegrity structures, with a focus establishing design, fabrication and erection guidelines, is an exciting prospect for future work.

References

[1] Robert E. Skelton and Mauricio C. de Oliveira. *Tensegrity Systems*. Springer Science, 2009.

[2] Mark Schenk. Statically balanced tensegrity mechanisms. Master's thesis, Delft University of Technology, 2005.

[3] John Rieffel, Francisco Valero-Cuevas, and Hod Lipson. Automated discovery and optimization of large irregular tensegrity structures. *Computers and Structures*, 2009.

[4] M. Pagitz and J.M. Mirats Tur. Finite element based form-finding algorithm for tensegrity structures. *International Journal of Solids and Structures*, 2009.

[5] Hoang Chi Tran and Jaehong Lee. Advanced form-finding of tensegrity structures. *Computers and Structures journal*, 2010.

[6] Herbert Klimke and Soeren Stephan. The making of a tensegrity tower.

[7] Bernard Adam and Ian F.C. Smith. Active tensegrity: A control framework for an adaptive civil-engineering structure. *Computers and Structures*, 2008.

[8] Giovani Gomez Estrada. *Analytical and numerical investigations of form-finding methods for tensegrity structures*. PhD thesis, Max-Planck-Institut fšur Metallforschung und Universitšat Stuttgart, 2007.

[9] Rene Motro. *Tensegrity. Structural Systems for the Future.* Hermes Science Publishing Limited, 2003.

[10] Anthony Pugh. *An Introduction to Tensegrity.* University of California Press, 1976.

[11] J.Y. Zhang, M. Ohsaki, and Y. Kanno. A direct approach to design of geometry and forces of tensegrity systems. *International Journal of Solids and Structures*, 2006.

[12] Marc Arsenault and Clement M. Gosselin. Kinematic and static analysis of a 3-pups spatial tensegrity mechanism. *Mechanism and Machine Theory*, 2009.

[13] Marc Arsenault and Clement M. Gosselin. Kinematic, static and dynamic analysis of a planar 2-dof spatial tensegrity mechanism. *Mechanism and Machine Theory*, 2006.

[14] Darrell Williamson, Robert E. Skelton, and Jeongheon Han. Equilibrium conditions of a tensegrity structure. *International Journal of Solids and Structures*, 2003.

[15] Bram de Jager and Robert E. Skelton. Stiffness of planar tensegrity truss topologies. *International Journal of Solids and Structures*, 2006.

[16] N. Bel Hadj Ali and I.F.C. Smith. Dynamic behavior and vibration control of a tensegrity structure. *International Journal of Solids and Structures*, 2010.

[17] Robert E. Skelton, Jean Paul Pinaud, and D.L. Mingori. Dynamics of the shell class of tensegrity structures. *Journal of the Franklin Institute*, 2001.

[18] Bernd Domer, Etienne Fest, Vikram Lalit, and Ian F. C. Smith. Combining dynamic relaxation method with artificial neural networks to

enhance simulation of tensegrity structures. *JOURNAL OF STRUCTURAL ENGINEERING*, 2003.

[19] Cornel Sultan, Martin Corless, and Robert E. Skelton. The prestressability problem of tensegrity structures: some analytical solutions. *International Journal of Solids and Structures*, 2001.

[20] William Brooks Whittier. Kinematic analysis of tensegrity structures. Master's thesis, Faculty of the Virginia Polytechnic Institute and State University, 2002.

[21] Hidenori Murakami. Static and dynamic analyses of tensegrity structures. part 1. nonlinear equations of motion. *International Journal of Solids and Structures*, 2001.

[22] Wang Bin Bing. *Free-standing Tension Structures: From tensegrity systems to cable-strut systems*. Spon Press, 2004.

[23] Cornel Sultan, Martin Corless, and Robert E. Skelton. The prestressability problem of tensegrity structures: some analytical solutions.

[24] A.G. Tibert and S. Pellegrino. Review of form-finding methods for tensegrity structures. *International Journal of Space Structures*, 2003.

[25] Yue Li, Xi-Qiao Feng, Yan-Ping Cao, and Huajian Gao. A monte carlo form-finding method for large scale regular and irregular tensegrity structures. *International Journal of Solids and Structures*, 2010.

[26] Robert Connelly. Rigidity and energy. *Inventiones Mathematicae*, 1982.

[27] R Connelly and M. XmeIl. Globally rigid symmetric tensegrities. *Structural Topology*, 1995.

[28] R. Burkhardt. The application of nonlinear programming to the design and validation of tensegrity structures with special attention to skew

prisms. *Journal of the International Association for Shell and Spatial Structures*, 2006.

[29] Sergi Hernandez Juan and Josep M. Mirats Tur. Tensegrity frameworks: Static analysis review. *Mechanism and Machine Theory*, 2008.

[30] *Form finding numerical methods for tensegrity systems*, 1994.

[31] Michael R. Barnes. Form finding and analysis of tension structures by dynamic relaxation. *International Journal of Space Structures*, 1999.

[32] *Data mining for design and manufacturing: methods and applications*. Kluwer Academic Publishers, 2001.

[33] Kurt Schmidheiny. Monte carlo experiments.

[34] B. Walsh. Markov chain monte carlo and gibbs sampling.

[35] G. Gomez Estrada, H.-J. Bungartz, and C. Mohrdieck. Numerical form-finding of tensegrity structures. *International Journal of Solids and Structures*, 2006.

[36] Julio Cesar Correa. Static analysis of tensegrity structures. Master's thesis, The Graduate School of the University of Florida, 2001.

[37] K. Kebiche, M.N. Kazi-Aoual, and R. Motro. Geometrical non-linear analysis of tensegrity systems. *Engineering Structures*, 1999.

[38] Cornel Sultan, Martin Corless, and Robert E. Skelton. Linear dynamics of tensegrity structures. *Engineering Structures*, 2002.

[39] F. Bossens, R.A de Callafon, and R.E. Skelton. Modal analysis of a tensegrity structure - an experimental study.

[40] Etienne Fest, Kristina Shea, Bernd Domer, , and Ian F. C. Smith. Adjustable tensegrity structures. *JOURNAL OF STRUCTURAL ENGINEERING*, 2003.

[41] Nikhil Vyas. Tensegrity based poultry shed. April 2009.

[42] Ian Ainsworth and Kathy Franklin. Kurilpa bridge. *The Arup Journal*, 2011.

[43] Robert William Burkhardt. *A Practical Guide to Tensegrity Design*. 2008.

[44] Robert E. Skelton, J. William Helton, Rajesh Adhikari, Jean-Paul Pinaud, and Waileung Chan. *Dynamics and Control of Aerospace Systems*, chapter An Introduction to the Mechanics of Tensegrity Structures. CRC Press LLC, 2002.

[45] Anders Sunde Wroldsen. *Modelling and Control of Tensegrity Structures*. PhD thesis, Department of Marine Technology. Norwegian University of Science and Technology, 2007.

[46] A. S. K. Kwan. A new approach to geometric nonlinearity of cable structures. *Computers and Structures*, 1998.

[47] Wanda J Lewis. *Tension Structures - Form and behaviour*. Thomas Telford Publishing, 2003.

[48] Sang-Eul Han and Kyoung-Su Lee. A study of the stabilizing process of unstable structures by dynamic relaxation method. *Computers and Structures*, 2003.

[49] D. S. Wakefield. Engineering analysis of tension structures: theory and practice. *Engineering Structures*, 1999.

[50] J. Quirant, M.N. Kazi-Aoual, and R. Motro. Designing tensegrity systems: the case of a double layer grid. *Engineering Structures*, 2003.

Appendix A

Tensegrity Designer Source Code

A.1 Vector Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Windows.Media.Media3D;

namespace Tensegrity_01
{
  class Vector
  {
    public double X;
    public double Y;
    public double Z;

    //Constructor
    public Vector(double Xval, double Yval, double Zval)
    {
```

```
  X = Xval;
  Y = Yval;
  Z = Zval;
}

// Constructor overload
public Vector()
{
  X = 0;
  Y = 0;
  Z = 0;
}

public string showAsString()
{
  double Mag = Magnitude();
  Mag = Math.Round(Mag, 3);
  double rX = Math.Round(X, 3);
  double rY = Math.Round(Y, 3);
  double rZ = Math.Round(Z, 3);
  return Mag + " {" + rX + " ; " + rY + " ; " + rZ + "}";
}

public string showCoordsAsString()
{
  double rX = Math.Round(X, 3);
  double rY = Math.Round(Y, 3);
  double rZ = Math.Round(Z, 3);
  return "( " + rX + " ; " + rY + " ; " + rZ + " )";
}

public double Magnitude()
{
  double mag = Math.Sqrt(X * X + Y * Y + Z * Z);
  return mag;
}
```

```
public void setAllComponents(double value)
{
  X = Y = Z = value;
}

public void MakeComponentsAbsolute()
{
  X = Math.Abs(X);
  Y = Math.Abs(Y);
  Z = Math.Abs(Z);
}

//return largest component
public double MaxComponent()
{
  double largest = Math.Max(X, Y);
  return Math.Max(largest, Z);
}

//divide each component of Vector by dividend
//return new Vector
public Vector divideByValue(double value)
{
  Vector newVector = new Vector();
  newVector.X = divideBy(X, value);
  newVector.Y = divideBy(Y, value);
  newVector.Z = divideBy(Z, value);
  return newVector;
}

public Vector multiplyByValue(double value)
{
  Vector newVector = new Vector();
  newVector.X = X * value;
  newVector.Y = Y * value;
  newVector.Z = Z * value;
  return newVector;
```

```
}

//add value to each component
public Vector addValue(double value)
{
  Vector newVector = new Vector();
  newVector.X = X + value;
  newVector.Y = Y + value;
  newVector.Z = Z + value;
  return newVector;
}

//add value to each component
public void addValueToThis(double value)
{
  X = X + value;
  Y = Y + value;
  Z = Z + value;
}

//vector addition - add to this vector, no return
public void addVectorToThis(Vector vector)
{
  X += vector.X;
  Y += vector.Y;
  Z += vector.Z;
}

// add and return new vector
public Vector addVector(Vector vector)
{
  Vector newVector = new Vector(X + vector.X, Y + vector.
     Y, Z + vector.Z);
  //newVector.X = X + vector.X;
  //newVector.Y = Y + vector.Y;
  //newVector.Z = Z + vector.Z;
  return newVector;
```

```
}

public Vector subtractVector(Vector vector)
{
  Vector newVector = new Vector();
  newVector.X = X - vector.X;
  newVector.Y = Y - vector.Y;
  newVector.Z = Z - vector.Z;
  return newVector;
}

public Vector multiplyByVector(Vector vector)
{
  Vector newVector = new Vector();
  newVector.X = X * vector.X;
  newVector.Y = Y * vector.Y;
  newVector.Z = Z * vector.Z;
  return newVector;
}

public Vector divideByVector(Vector vector)
{
  Vector newVector = new Vector();
  newVector.X = divideBy(X, vector.X);
  newVector.Y = divideBy(Y, vector.Y);
  newVector.Z = divideBy(Z, vector.Z);
  return newVector;
}

//division function that returns 0 if division by zero
private double divideBy(double dividend, double divisor)
{
  if (divisor == 0)
  {
    return 0;
  }
  else
```

```
  {
    return dividend / divisor;
  }
}

// returns vector signs for each component
public Vector SignOfVector()
{
  return new Vector(GetSign(X), GetSign(Y), GetSign(Z));
}

public Vector ReverseSigns()
{
  return new Vector(-X, -Y, -Z);
}

public double GetSign(double val)
{
  if (val == 0)
  {
    return 1;
  }
  else if (val < 0)
  {
    return -1;
  }
  else
  {
    return 1;
  }
}

public void MakeUnitVector()
{
  double Mag = Magnitude();
  X /= Mag;
  Y /= Mag;
```

```
      Z /= Mag;
    }

    public Vector UnitVector()
    {
      double Mag = Magnitude();
      return new Vector(X/Mag, Y/Mag, Z/Mag);
    }

    public Vector ProjectVectors(Vector FromThisVector,
       Vector ToThisVector)
    {
      Vector ToDirection = ToThisVector.UnitVector();

      Vector3D V1 = new Vector3D(FromThisVector.X,
         FromThisVector.Y, FromThisVector.Z);
      Vector3D V2 = new Vector3D(ToDirection.X, ToDirection.Y
         , ToDirection.Z);

      double dot = Vector3D.DotProduct(V1, V2);

      //dot = Math.Abs(dot);
      return ToDirection.multiplyByValue(dot);
    }
  }
}
```

A.2 Node Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Tensegrity_01
{
  class Node : Vector
  {
    public int nodeNumber;
    public string nodeLabel;

    //attributes
    public double mass;
    public double timeIncrement;
    public double lambda; //same as for global structure

    //public double stiffness;
    public Vector Stiffness = new Vector();

    public Vector Fixity = new Vector(); // 0 = free, 1 =
       fixed (only displacements, NOT rotations)

    public Vector ForceApplied = new Vector(); // External
       force APPLIED to node
    public double appliedForceFactor = 1; //factor for
       scaling the applied force

    //Make these functions
    public Vector ForceExternal = new Vector(); // Selfweight
    public Vector ForceInternal = new Vector(); // Applied on
        node via member force
    public Vector ForceResidual = new Vector();
```

```
public Vector Acceleration = new Vector();
public Vector Velocity = new Vector();
public Vector VelocityPrevious = new Vector();

public Vector Displacement = new Vector();

//Limiting
const double MAX_DISPLACEMENT = 0.050; // (mm) the
   maximum allowable displacement in any direction in a
   timestep.
const double MIN_STIFFNESS = 100000;
const double MAX_RESIDUAL = 50;

public Node()
{
}

public Node(double nX, double nY, double nZ)
{
  X = nX;
  Y = nY;
  Z = nZ;
}

// clear mass, stiffness, forces
public void Clear()
{
  mass = 0F;
  Stiffness = new Vector();
  ForceExternal = new Vector();
  ForceInternal = new Vector();
}

public void Calculate(double timeStep, double theLambda)
{
  timeIncrement = timeStep;
  lambda = theLambda;
```

```
  //calcForceInternal();  //not needed
  CalcForceResidual();
  CalcVelocity();
  //CalcAcceleration(); // not using F=ma
  CalcDisplacement();
  //calcPosition();
}

// Force applied to node via member
// Should be resolved onto member
public void CalcForceInternal()
{
  // Internal Force = K * d
  ForceInternal = Displacement.multiplyByVector(Stiffness
     );
}

public void CalcForceResidual()
{
  Vector PrevResidual = ForceResidual.multiplyByValue(1);
  // R = External Forces - Internal Forces

  // Fapplied + Fext + Ma - Finternal

  //Vector Fma = new Vector();
  //Fma = acceleration.multiplyByValue(mass);

  //Vector Ftotal = new Vector();
  //Ftotal = Ftotal.addVector(forceApplied).addVector(
     forceExternal).subtractVector(Fma).subtractVector(
     forceInternal);
  ForceResidual = ForceApplied.multiplyByValue(
     appliedForceFactor).addVector(ForceExternal).
     addVector(ForceInternal);

  //return Ftotal;
```

```
  //Restrain where required
  if (Fixity.X == 1)
  {
    ForceResidual.X = 0;
  }

  if (Fixity.Y == 1)
  {
    ForceResidual.Y = 0;
  }

  if (Fixity.Z == 1)
  {
    ForceResidual.Z = 0;
  }

  double resMag = ForceResidual.Magnitude();

  double resMagPrev = PrevResidual.Magnitude();

  double magDiff = ForceResidual.X - PrevResidual.X; //
     resMag - resMagPrev;
}

public void CalcVelocity()
{
  //store old velocity
  VelocityPrevious = new Vector().addVector(Velocity);

  // v2 = v1 + F * t / M
  // v2 = v1 + 2R/K ( t = 1)
  // tmax = sqrt( 2 * M / K )

  //try
  double nodeLambda = lambda;
```

```
  // if t = 1 then M = K / 2
  //Vector Mass = Stiffness.multiplyByValue(timeIncrement
      * timeIncrement).divideByValue(2);
  Vector Mass = new Vector();
  //Mass.setAllComponents(mass);

  Mass.setAllComponents(nodeLambda * timeIncrement *
     timeIncrement / 2);
  Mass = Mass.multiplyByVector(Stiffness);

  Vector newVel = new Vector();
  newVel = Velocity.addVector(ForceResidual.
     multiplyByValue(timeIncrement).divideByVector(Mass))
     ;

  //newVel = Velocity.addVector(ForceResidual.
     multiplyByValue(2).divideByVector(Stiffness));
  Velocity = newVel;
}

//find new positions of nodes due to displacement
public void SetNewPosition()
{
  this.addVectorToThis(Displacement);
}

public void CalcDisplacement()
{
  // s2 = s1 + v2*timeIncrement
  Vector newDispl = new Vector();
  Vector displChange = Velocity.multiplyByValue(
     timeIncrement);

  //limit the max displacement to prevent the system from
      displacing uncontrollably
  displChange = LimitDisplacement(displChange);
```

```
  newDispl = Displacement.addVector(displChange);

  int decimals = 6;
  newDispl.X = Math.Round(newDispl.X, decimals);
  newDispl.Y = Math.Round(newDispl.Y, decimals);
  newDispl.Z = Math.Round(newDispl.Z, decimals);

  Displacement = newDispl;
}

private Vector LimitDisplacement(Vector displChange)
{
  double displMag = displChange.Magnitude();
  if (displMag > MAX_DISPLACEMENT)
  {
    displChange.MakeUnitVector();
    displChange = displChange.multiplyByValue(
       MAX_DISPLACEMENT);
  }
  displMag = displChange.Magnitude();

  return displChange;
}

// correct displacements to that at KE peak
public void CalcDisplacementAtKineticEnergyPeak(double
   beta)
{
  //Displacement now equals wrong displacement so we
     correct it, i.e. s = s2

  Vector CorrectedDispl = new Vector();

  // s2-corr = s2 - v2 * t - beta * t * v1
  Vector displDiff = Velocity.multiplyByValue(
     timeIncrement).subtractVector(VelocityPrevious.
     multiplyByValue(beta * timeIncrement));
```

```
      displDiff = LimitDisplacement(displDiff);
      CorrectedDispl = Displacement.subtractVector(displDiff)
          ;

      // s2-corr = s2 - (1-beta)*v1 - 2F/K
      // TimeIncrement = 1 second
      //CorrectedDispl = Displacement.subtractVector(
          VelocityPrevious.multiplyByValue(1 + beta));
      //CorrectedDispl = CorrectedDispl.subtractVector(
          ForceResidual.multiplyByValue(2).divideByVector(
          Stiffness));

      Displacement = CorrectedDispl;
    }

    public double CalcKineticEnergy()
    {
      // KE = 1/2 * m * v^2
      double v = Velocity.Magnitude();
      return mass / 2 * v * v;
    }
  }
}
```

A.3 Member Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Windows.Media.Media3D;

namespace Tensegrity_01
{
  class Member
  {

    public int memberNumber;
    public string memberLabel;

    public Node StartNode;
    public Node EndNode;
    public MemberType memberType = new MemberType(); //make
       this custom type (enum)
    public Vector OriginalLength; // StartNode - EndNode (
       excl. sign)
    public double originalCableLengthFactor = 1; //Factor for
        scaling original length (for tension/slack in cables
       ...CABLES only)
    public Vector CurrentLength = new Vector();
    public Vector TotalDeflection = new Vector();
    public Vector Stiffness = new Vector();
    public double mass;
    public double customLength = 0;

    public Vector EStiffness = new Vector(); //elastic
       stiffness: EA/L
    public Vector GStiffness = new Vector(); //geometric
       stiffness: T/L
```

```
public double totalStiffness;

public Vector Selfweight = new Vector();

Vector CurStartNode = new Vector();
Vector CurEndNode = new Vector();

Vector StartNodeDirection = new Vector();
Vector EndNodeDirection = new Vector();

public Vector ForceOnStartNode = new Vector();
public Vector ForceOnEndNode = new Vector();

public Vector MemberForce = new Vector(); //
   StiffnessForce + ...
public Vector NodalForce = new Vector(); // Force from
   nodes on member
public Vector StiffnessForce = new Vector();
public Vector TensionForce = new Vector();

//resolved along member
public double memberDeflection;
public double memberStiffnessForce;

//constructor
public Member(Node StartNode0, Node EndNode0, MemberType
   MemType)
{
  StartNode = StartNode0;
  EndNode = EndNode0;
  memberType = MemType;

  CalcOriginalLength();
  CalcCurrentLength();
}

public void CalcOriginalLength()
```

```
{
  OriginalLength = StartNode.subtractVector(EndNode);
  OriginalLength.MakeComponentsAbsolute();

  // apply initial deformation
  if (customLength != 0)
  {
    OriginalLength.MakeUnitVector();
    OriginalLength = OriginalLength.multiplyByValue(
        customLength);
  }

  //apply slackness factor
  if (memberType.typeNumber == 1) //cable
  {
    OriginalLength = OriginalLength.multiplyByValue(
        originalCableLengthFactor);
  }
}

//constructor overload
public Member()
{

}

//calculate attributes for this step
public void Calculate()
{
  CalcNodeDirections();
  CalcCurrentLength();
  CalcTotalDeflection();
  CalcSelfweight();
  //CalcNodeDirections();
  CalcMemberForce();
}
```

```
//resolve all attributes calculated below
public void resolveAll()
{
  resolveMass();
  resolveStiffness();
  //resolveSelfweight();
  ResolveForce();
}

// Calcs the direction vectors to the nodes from the
   midpt of the member
// i.e. direction to node along member
// used to resolve member forces to nodes and vice versa
// needed for sign conventions
// Note the reversed directions of vectors (i.e. NOT
   EndNode minus StartNode as usual)
public void CalcNodeDirections()
{
  CurStartNode = StartNode.addVector(StartNode.
     Displacement);
  CurEndNode = EndNode.addVector(EndNode.Displacement);

  Vector MidPoint = CurStartNode.addVector(CurEndNode).
     divideByValue(2);
  StartNodeDirection = MidPoint.subtractVector(
     CurStartNode);
  EndNodeDirection = MidPoint.subtractVector(CurEndNode);

  StartNodeDirection.MakeUnitVector();
  EndNodeDirection.MakeUnitVector();
}

public void CalcCurrentLength()
{
  Vector NewLength = CurStartNode.subtractVector(
     CurEndNode);
  NewLength.MakeComponentsAbsolute();
```

```
  //NewLength = NewLength.subtractVector(StartNode.
     displacement).subtractVector(EndNode.displacement);
  //NewLength.MakeComponentsAbsolute();
  //return NewLength;

  CurrentLength = NewLength;
}
private double SignOfLengthChange()
{
  double lengthChange = ChangeInLength();
  return new Vector().GetSign(lengthChange);
}

public void CalcSelfweight()
{
  //only in y direction, i.e. down
  double sWeight = mass * 9.81; //Newtons

  //downwards = negative y direction
  sWeight *= -1;
  Selfweight.Y = sWeight;

  //return selfweight;
}

public void CalcTotalDeflection()
{
  TotalDeflection = CurrentLength.subtractVector(
    OriginalLength);
  double mag = TotalDeflection.Magnitude();
  memberDeflection = new Vector().ProjectVectors(
     TotalDeflection, StartNodeDirection).Magnitude();
  memberDeflection *= SignOfLengthChange();
}

public void SetTension(double tensionMag)
{
```

```
  CalcNodeDirections();
  TensionForce = EndNodeDirection.multiplyByValue(
     tensionMag);
}

public void SetLength(double newLength)
{
  //CalcNodeDirections();
  //CalcCurrentLength();

  customLength = newLength;
  CalcOriginalLength();
}

public void SetLengthFactor(double prestressFactor) //< 1
    means tension, > 1 means slack
{
  CalcNodeDirections();
  CalcCurrentLength();
  double newLength = CurrentLength.Magnitude() *
     prestressFactor;
  SetLength(newLength);
}

//internal member force in this member
//currently only stiffness force
// Not taking mass force into account ryt now
public void CalcMemberForce()
{
  //selfweight + external applied force + internal
     applied prestress
  //double selfweight = Selfweight();
  //return new Vector(0, selfweight, 0);

  //CalcNodalForce();
  CalcStiffnessForce();
```

```
  //MemberForce = NodalForce.addVector(StiffnessForce).
     addVector(TensionForce); // + .....
  //MemberForce = MemberForce.subtractVector(NodalForce);

  MemberForce = new Vector();
  MemberForce = MemberForce.addVector(StiffnessForce); //
      + .....
}

// if tensegrity system then apply member type resisting
   restraints
// i.e. cables -> only tension
//    struts -> only compression
public void FixMemberForce()
{
  double forceMag = MemberForce.Magnitude();
  double lengthChange = ChangeInLength();

  // Check whether strut or cable and set to zero if
     wrong sign
  if (memberType.typeNumber == 1)
  { //cable
    if (lengthChange < 0) // stretched
    {
      MemberForce.setAllComponents(0);
    }
  }
  else
  { //strut
    if (lengthChange > 0)
    {
      //MemberForce.setAllComponents(0);
    }
  }
}

public double ChangeInLength()
```

```
{
  return CurrentLength.Magnitude() - OriginalLength.
     Magnitude();
}

// Get the force due to both stiffnesses
public void CalcStiffnessForce()
{
  // Calculate stiffness along member
  memberStiffnessForce = memberDeflection *
     totalStiffness;
  // resolve onto stiffness vector
  Vector MemStiffnessForce = StartNodeDirection.
     multiplyByValue(memberStiffnessForce);
  StiffnessForce = MemStiffnessForce;

  //StiffnessForce = new Vector().ProjectVectors(
     MemStiffnessForce, StartNodeDirection);
}

public void CalcNodalForce()
{
  Vector StartNodeForce = StartNode.ForceApplied.
     addVector(StartNode.ForceExternal);
  Vector EndNodeForce = EndNode.ForceApplied.addVector(
     EndNode.ForceExternal);

  // resolve directions (and sort out sign convention)
  StartNodeForce = StartNodeForce.multiplyByVector(
     EndNodeDirection);
  EndNodeForce = EndNodeForce.multiplyByVector(
     StartNodeDirection);

  NodalForce = StartNodeForce.addVector(EndNodeForce);
}

// Resolve member internal force onto nodes
```

```
public void ResolveForce()
{
  Vector Force = MemberForce;

  double forceMag = Force.Magnitude() *
     SignOfLengthChange();
  //forceMag = SignOfLengthChange();

  ForceOnStartNode = StartNodeDirection.multiplyByValue(
     forceMag);
  ForceOnEndNode = EndNodeDirection.multiplyByValue(
     forceMag);

  StartNode.ForceInternal.addVectorToThis(
     ForceOnStartNode);
  EndNode.ForceInternal.addVectorToThis(ForceOnEndNode);
}

//resolve mass onto start and end nodes (1/2 1/2)
public void resolveMass()
{
  mass = OriginalLength.Magnitude() * memberType.Area *
     memberType.density;
  StartNode.mass += mass / 2;
  EndNode.mass += mass / 2;
}

public void resolveStiffness()
{
  //EStiffness =
  double eStiffness = memberType.E * memberType.Area /
     OriginalLength.Magnitude();
  //double eee = EStiffness.Magnitude();

  double gStiffness = MemberForce.Magnitude() /
     CurrentLength.Magnitude();
  gStiffness = Math.Abs(gStiffness);
```

```
        eStiffness /= 2;
        totalStiffness = eStiffness + gStiffness;

        Vector StartNodeStiffness = new Vector();
        Vector EndNodeStiffness = new Vector();

        //set all components the same
        StartNodeStiffness.setAllComponents(totalStiffness);
        EndNodeStiffness.setAllComponents(totalStiffness);

        StartNode.Stiffness.addVectorToThis(StartNodeStiffness)
            ;
        EndNode.Stiffness.addVectorToThis(EndNodeStiffness);

      }

      public void resolveSelfweight()
      {
        //double selfweight = Selfweight();

        //keep mass constant
        if (StartNode.ForceExternal.Y == 0)
        {
          StartNode.ForceExternal.Y += Selfweight.Y / 2;
          EndNode.ForceExternal.Y += Selfweight.Y / 2;
        }
      }
    }
}
```

A.4 MemberType Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Tensegrity_01
{
  class MemberType
  {
    public int typeNumber;    // 1 = Cable, 2 = Strut
    public string name;

    public double E; // Pa
    public double density; // kg/m3
    public double radius; //m
    public double boreRadius = 0; //m

    double _area = 0;

    public MemberType()
    {
      density = 1000;
    }

    public double Area
    {
      get
      {
        if (_area == 0)
        {
          calcArea();
        }

        return _area;
```

```
      }
      set
      {
        _area = value;
      }
    }

    public void calcArea()
    {
      _area = Math.PI * (radius * radius - boreRadius *
          boreRadius);
    }
  }
}
```

A.5 Tensegrity Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Windows.Forms;
using System.Windows.Media;
using System.Windows.Media.Media3D;
using System.IO;

namespace Tensegrity_01
{
  class Tensegrity
  {
    //for drawing
    _3D_Drawing_Control.UserControl1 drawing;
    public int drawingScaleFactor = 1;
    public int selectedNodeNum;
    public int selectedMemberNum;

    public string name;

    int iNodeCount = 0;
    int iMemberCount = 0;
    int iMemberTypeCount = 0;

    public int activeNode; //the number of the node that will
        be monitored and F vs D recorded

    public Node[] nodes = new Node[1];
    public Member[] members = new Member[1];
    public MemberType[] memberTypes = new MemberType[1];

    public Node[] nodesStored = new Node[100];
```

```
public double kineticEnergyStored = 0;

public double timeIncrement = 1;
public double lambda; //used to adjust mass for optimum
   convergence
public double lambdaInitial = 100;
public double lambdaFactor = 100;
public double lambdaIncreaseFactor = 1.5; //1.02;
public int lamdaUseCount = 5; // after how many peaks
   should we increase lambda

public double kineticEnergy = 0;
public double kineticEnergyAvg = 0;
public double kineticEnergyPrevious = 0;
public double kineticEnergyMax = 0;
public double kineticEnergyLimit = 10E6;//the maximum
   allowed KE before analysis stopped
public int peakCount = 0;

public int iterations = 0; //number of iterations
public bool kineticDamping = true; // damp system
public bool tensegritySystem = true; // is this a single
   load direction resisting member system

//for kinematic damping
const int MIN_ITERATIONS_BEFORE_DAMPING = 10; //the
   minimum number of iterations to be completed before
   kinetic damping is allowed
public int lastDampedIterationNum = 0;
public double kE1 = 0;
public double kE2 = 0;
public double kE3 = 0;

//Member types
public MemberType Cable;
public MemberType Strut;
```

```
//constructor
public Tensegrity()
{
  //set above
  //timeIncrement = 1E-2;

  //cable
  Cable = addMemberType("cable");
  //Cable.radius = 0.015;
  Cable.radius = 0.0005; //1mm diametre i.e. 0.5mm radius
  Cable.E = 193E9; //stainless steel
  //Cable.E /= 10; //testing

  //strut
  Strut = addMemberType("strut");
  //Strut.radius = 0.05;
  Strut.radius = 0.005; //10mm diameter
  Strut.boreRadius = 0.0037; // 1.3mm wall thickness
  Strut.E = 69E9; //aluminium

  lambda = lambdaInitial;
}

// add a node to the system
public MemberType addMemberType(string typeName)
{
  iMemberTypeCount++; //dont need this, can just use
     memberTypes.Length
  //start numbering from 1 and NOT 0 (array index)
  MemberType[] mtNew = new MemberType[iMemberTypeCount +
     1]; //make new array with 1 extra node
  Array.Copy(memberTypes, mtNew, memberTypes.Length); //
     copy old nodes to new array

  //add new MemberType to new array
  memberTypes = mtNew; //set original to new array
```

```
  MemberType newMT = new MemberType();
  newMT.typeNumber = iMemberTypeCount;
  newMT.name = typeName;

  //set attributes


  memberTypes[iMemberTypeCount] = newMT;

  return memberTypes[iMemberTypeCount];
}

public void Run()
{
  iterations++;
  resolveMembersToNodes();
  calcNodes();
}

// add a node to the system
public Node addNode(Point3D nodePoint, Vector Fixity)
{
  iNodeCount++;
  //start numbering from 1 and NOT 0 (array index)
  Node[] nodesNew = new Node[iNodeCount + 1]; //make new
     array with 1 extra node
  Array.Copy(nodes, nodesNew, nodes.Length); //copy old
     nodes to new array
  //add new node to new array
  nodes = nodesNew; //set original to new array

  Node newNode = new Node();
  newNode.nodeNumber = iNodeCount;
  newNode.X = nodePoint.X;
  newNode.Y = nodePoint.Y;
  newNode.Z = nodePoint.Z;
```

```
  newNode.Fixity = Fixity; //support conditions

  nodes[iNodeCount] = newNode;

  return nodes[iNodeCount];
}

// overload method
public Node addNode(Point3D nodePoint)
{
  return addNode(nodePoint, new Vector(0, 0, 0));
}

// overload method
public Node addNode(double X, double Y, double Z)
{
  return addNode(new Point3D(X, Y, Z));
}

// find the node and return it, or create new node if
   none exists on that location
public Node GetNode(Point3D nodePoint)
{
  for (int i = 1; i < nodes.Length; i++)
  {
    Node n = nodes[i];

    if (n.X == nodePoint.X && n.Y == nodePoint.Y && n.Z
      == nodePoint.Z)
    {
      // node exists on this point
      return n;
    }
  }

  return addNode(nodePoint);
}
```

```
// overload
public Node GetNode(double X, double Y, double Z)
{
  return GetNode(new Point3D(X, Y, Z));
}

public Member addMember(int StartNodeNum, int EndNodeNum,
    int memberTypeNum)
{
  iMemberCount++; //dont need this, can just use members.
     Length
  //start numbering from 1 and NOT 0 (array index)
  Member[] membersNew = new Member[iMemberCount + 1]; //
     make new array with 1 extra node
  Array.Copy(members, membersNew, members.Length); //copy
      old nodes to new array
  //add new node to new array
  members = membersNew; //set original to new array

  Member newMember = new Member(nodes[StartNodeNum],
     nodes[EndNodeNum], memberTypes[memberTypeNum]);
  newMember.memberNumber = iMemberCount;

  members[iMemberCount] = newMember;
  return members[iMemberCount];
}

public Member addMember(Node StartNode0, Node EndNode0,
   int memberTypeNum)
{
  return addMember(StartNode0.nodeNumber, EndNode0.
     nodeNumber, memberTypeNum);
}

//returns component of length in particular direction, 1=
   x, 2=y, etc.
```

```
public double memberLength(int memberNum, int direction)
{
  double startCoord = members[memberNum].StartNode.X; //
     direction 1
  double endCoord = members[memberNum].EndNode.X;
  return startCoord - endCoord;
}

public void setMemberLength(string memLabel, double
   memLength)
{
  for (int i = 1; i < members.Length; i++)
  {
    if (members[i].memberLabel == memLabel)
    {
      members[i].SetLength(memLength);
    }
  }
}

//overload
public void SetCableLengthFactor(double cableLengthFactor
   )
{
  SetCableLengthFactor(cableLengthFactor, "all");
}

public void SetCableLengthFactor(double cableLengthFactor
   , string memLabel)
{
  for (int i = 1; i < members.Length; i++)
  {
    if (members[i].memberLabel == memLabel || memLabel ==
        "all")
    {
      members[i].originalCableLengthFactor =
         cableLengthFactor;
```

```
      //this function checks if it is a cable
      members[i].CalcOriginalLength();
    }
  }
}

public void SetNodesAppliedForceFactor(double
    appliedForceFactor)
{
  for (int i = 1; i < nodes.Length; i++)
  {
    nodes[i].appliedForceFactor = appliedForceFactor;
  }
}

public void setNodeAppliedForce(string nLabel, Vector
    nAppliedForce)
{
  for (int i = 1; i < nodes.Length; i++)
  {
    if (nodes[i].nodeLabel == nLabel)
    {
      nodes[i].ForceApplied = nAppliedForce;
    }
  }
}

public void setNodeAppliedVerticalForce(string nLabel,
    double vForce)
{
  setNodeAppliedForce(nLabel, new Vector(0, vForce, 0));
}

public void setTotalAppliedVerticalForce(double
    vTotalForce)
{
```

```
  double nodeCount = 0;

  for (int i = 1; i < nodes.Length; i++)
  {
    if (nodes[i].nodeLabel == "top")
    {
      nodeCount++;
    }
  }

  setNodeAppliedVerticalForce("top", vTotalForce /
     nodeCount);
}

public void setNodeDisplacement(string nLabel, double
   vDispl)
{
  setNodeDisplacement(nLabel, new Vector(0, vDispl, 0));
}

public void setNodeDisplacement(string nLabel, Vector
   nDispl)
{
  for (int i = 1; i < nodes.Length; i++)
  {
    if (nodes[i].nodeLabel == nLabel)
    {
      nodes[i].Displacement = nDispl;
    }
  }
}

// Resolves member forces and stiffnesses onto nodes for
   calculation
public void resolveMembersToNodes()
{
  //clear previous values
```

```
  clearNodes();

  //foreach (Member m in members)
  for (int i = 1; i < members.Length; i++)
  {
    members[i].Calculate();

    if (tensegritySystem)
    {
      members[i].FixMemberForce();
    }

    members[i].resolveAll();
    //m.resolveAll();
  }
}

//clear mass, stiffness and selfweight before resolving
   members
public void clearNodes()
{
  for (int i = 1; i < nodes.Length; i++)
  {
    nodes[i].Clear();
  }
}

public void AddConfigToTreeview(TreeView tvw)
{
  tvw.Nodes.Clear();
  TreeNode tnModelNodes; // main nodes


  tnModelNodes = tvw.Nodes.Add("Model");
  tnModelNodes.Nodes.Add("Name", "Model Name");
  tnModelNodes.Nodes.Add("Time", "Time Increment").Nodes.
     Add(timeIncrement.ToString());
```

```
tnModelNodes.Nodes.Add("Lambda", "Lambda").Nodes.Add(
   lambda.ToString());
tnModelNodes.ExpandAll();

// Nodes
tnModelNodes = tvw.Nodes.Add("Nodes");

for (int i = 1; i < nodes.Length; i++)
{
  Node n = nodes[i];
  TreeNode tnNodes = tnModelNodes.Nodes.Add(n.
     nodeNumber.ToString(), "Node " + n.nodeNumber);

  //treenode ref for variables
  TreeNode tnNodeVar;

  //co-ordinates
  tnNodeVar = tnNodes.Nodes.Add("Coordinates", "Co-
     ordinates " + n.showCoordsAsString());
  tnNodeVar.Nodes.Add("X", n.X.ToString());
  tnNodeVar.Nodes.Add("Y", n.Y.ToString());
  tnNodeVar.Nodes.Add("Z", n.Z.ToString());

  //Fixity
  tnNodeVar = tnNodes.Nodes.Add("Fixity", "Fixity " + n
     .Fixity.showCoordsAsString());
  tnNodeVar.Nodes.Add("X", n.Fixity.X.ToString());
  tnNodeVar.Nodes.Add("Y", n.Fixity.Y.ToString());
  tnNodeVar.Nodes.Add("Z", n.Fixity.Z.ToString());

  //ForceApplied
  tnNodeVar = tnNodes.Nodes.Add("Force Applied", "Force
      Applied " + n.ForceApplied.showCoordsAsString());
  tnNodeVar.Nodes.Add("X", n.ForceApplied.X.ToString())
     ;
  tnNodeVar.Nodes.Add("Y", n.ForceApplied.Y.ToString())
     ;
```

```
tnNodeVar.Nodes.Add("Z", n.ForceApplied.Z.ToString())
    ;

//ForceInternal
tnNodeVar = tnNodes.Nodes.Add("Force Internal", "
    Force Internal " + n.ForceInternal.
    showCoordsAsString());
tnNodeVar.Nodes.Add("X", n.ForceInternal.X.ToString()
    );
tnNodeVar.Nodes.Add("Y", n.ForceInternal.Y.ToString()
    );
tnNodeVar.Nodes.Add("Z", n.ForceInternal.Z.ToString()
    );

//ForceResidual
tnNodeVar = tnNodes.Nodes.Add("Force Residual", "
    Force Residual " + n.ForceResidual.
    showCoordsAsString());
tnNodeVar.Nodes.Add("X", n.ForceResidual.X.ToString()
    );
tnNodeVar.Nodes.Add("Y", n.ForceResidual.Y.ToString()
    );
tnNodeVar.Nodes.Add("Z", n.ForceResidual.Z.ToString()
    );

//Displacement
tnNodeVar = tnNodes.Nodes.Add("Displacement", "
    Displacement " + n.Displacement.showCoordsAsString
    ());
tnNodeVar.Nodes.Add("X", n.Displacement.X.ToString())
    ;
tnNodeVar.Nodes.Add("Y", n.Displacement.Y.ToString())
    ;
tnNodeVar.Nodes.Add("Z", n.Displacement.Z.ToString())
    ;

//Mass
```

```
  tnNodeVar = tnNodes.Nodes.Add("Mass", "Mass: " + n.
     mass.ToString() + " kg");

  //tnNodes.Expand();

  //tnModelNodes.Expand();
}

// Members
tnModelNodes = tvw.Nodes.Add("Members");
double cableLength = 0;
double cableCount = 0;

for (int i = 1; i < members.Length; i++)
{
  Member m = members[i];
  TreeNode tnNodes = tnModelNodes.Nodes.Add(m.
     memberNumber.ToString(), "Member " + m.
     memberNumber + " ( " + m.MemberForce.Magnitude().
     ToString() + " N )");

  //treenode ref for variables
  TreeNode tnNodeVar;

  //StartNode
  tnNodeVar = tnNodes.Nodes.Add("StartNode", "Start
     Node " + m.StartNode.showCoordsAsString());
  tnNodeVar.Nodes.Add("X", m.StartNode.X.ToString());
  tnNodeVar.Nodes.Add("Y", m.StartNode.Y.ToString());
  tnNodeVar.Nodes.Add("Z", m.StartNode.Z.ToString());

  //EndNode
  tnNodeVar = tnNodes.Nodes.Add("EndNode", "End Node "
     + m.EndNode.showCoordsAsString());
  tnNodeVar.Nodes.Add("X", m.EndNode.X.ToString());
  tnNodeVar.Nodes.Add("Y", m.EndNode.Y.ToString());
  tnNodeVar.Nodes.Add("Z", m.EndNode.Z.ToString());
```

```
  //Attributes
  tnNodeVar = tnNodes.Nodes.Add("Attributes", "
     Attributes");
  tnNodeVar.Nodes.Add("Original Length", "Original
     Length: " + m.OriginalLength.showAsString());
  tnNodeVar.Nodes.Add("Current Length", "Current Length
     : " + m.CurrentLength.showAsString());
  tnNodeVar.Nodes.Add("Stiffness", "Stiffness: " + m.
     Stiffness.showAsString());

  //Forces
  tnNodeVar = tnNodes.Nodes.Add("Forces", "Forces");
  tnNodeVar.Nodes.Add("Total Member Force", "Total
     Member Force: " + m.MemberForce.showAsString());
  tnNodeVar.Nodes.Add("Nodal Force", "Nodal Force: " +
     m.NodalForce.showAsString());
  tnNodeVar.Nodes.Add("Stiffness Force", "Stiffness
     Force: " + m.StiffnessForce.showAsString());
  tnNodeVar.Nodes.Add("Tension Force", "Tension Force:
     " + m.TensionForce.showAsString());

  if (m.memberType.name == "cable")
  {
    cableLength += m.OriginalLength.Magnitude();
    cableCount++;
  }

  //tnNodes.Expand();
}

tvw.Nodes[0].Nodes.Add("CableCount", "No. of Cables").
   Nodes.Add(cableCount.ToString());
tvw.Nodes[0].Nodes.Add("CableLength", "Total Cable
   Length").Nodes.Add(cableLength.ToString());

//scroll to top
```

```
  tvw.SelectedNode = tvw.Nodes[0];

  //expand main nodes
  //tnModelNodes.Expand();
}

public void StoreKineticEnergyData(ListBox lst, System.IO
    .StreamWriter Writer)
{
  double kE = Math.Round(kineticEnergy, 5);
  //lst.Items.Insert(0, (lst.Items.Count + 1) + ". " + kE
      + " J");

  //store kinetic energy in file
  Writer.WriteLine(iterations.ToString() + "," +
     kineticEnergy.ToString()
    + "," + lambda); // + "," + lambda/100000 );

  if (kineticEnergyMax != 0)
  {
    //lst.Items.Insert(0, "     Max KE: " +
       kineticEnergyMax);
    kineticEnergyMax = 0; //dont add multiple times
  }
}

public void addNodesToListview(ListView lvw)
{
  //lvw.Items.Clear();
  lvw.Items.Add(iterations + " KE: " + kineticEnergy + "
     J");

  for (int i = 1; i < nodes.Length; i++)
  {
    Node n = nodes[i];
    ListViewItem lvi = lvw.Items.Add(n.nodeNumber.
       ToString());
```

```
    lvi.SubItems.Add(n.showAsString());
    lvi.SubItems.Add(n.Stiffness.showAsString());
    //lvi.SubItems.Add(n.stiffness.ToString());
    //lvi.SubItems.Add(n.forceExternal.showAsString());
    lvi.SubItems.Add(n.ForceApplied.showAsString());
    lvi.SubItems.Add(n.ForceInternal.showAsString());
    lvi.SubItems.Add(n.ForceExternal.showAsString());
    lvi.SubItems.Add(n.ForceResidual.showAsString());
    lvi.SubItems.Add(n.Acceleration.showAsString());
    //lvi.SubItems.Add("N/A");
    lvi.SubItems.Add(n.Velocity.showAsString());
    lvi.SubItems.Add(n.Displacement.showAsString());
  }
}

public void addMembersToListview(ListView lvw)
{
  //lvw.Items.Clear();
  lvw.Items.Add("------");

  for (int i = 1; i < members.Length; i++)
  {
    Member m = members[i];
    ListViewItem lvi = lvw.Items.Add(m.memberNumber.
        ToString());
    lvi.SubItems.Add(m.StartNode.nodeNumber.ToString());
    lvi.SubItems.Add(m.EndNode.nodeNumber.ToString());
    //lvi.SubItems.Add(m.length3D.ToString());
    lvi.SubItems.Add(m.CurrentLength.Magnitude().ToString
        ());
    lvi.SubItems.Add(m.OriginalLength.Magnitude().
        ToString());
    lvi.SubItems.Add(m.totalStiffness.ToString());
    lvi.SubItems.Add(m.NodalForce.showAsString());
    lvi.SubItems.Add(m.StiffnessForce.showAsString());
    lvi.SubItems.Add(m.MemberForce.showAsString());
    //lvi.SubItems.Add(m.TotalDeflection.showAsString());
```

```
    double lenChange = Math.Round(m.ChangeInLength(), 3);
    double memDefl = Math.Round(m.memberDeflection, 3);
    lvi.SubItems.Add(lenChange + " , " + memDefl);

    lvi.SubItems.Add(m.GStiffness.showAsString());

    //double area = m.memberType.Area();

    string details = "E = " + m.memberType.E + "GPa | A =
        " + m.memberType.Area +
              "m3 | dens. = " + m.memberType.density + "
                 kg/m3";
    //lvi.SubItems.Add(details);
  }
}

//calculate current config.stuff - acc, vel, displ. etc.
public void calcNodes()
{
  //calc average
  kineticEnergyAvg = (kE1 + kE2 + kE3) / 3;

  //reset energy
  kE1 = kineticEnergyPrevious;
  kE2 = kineticEnergy;

  kineticEnergyPrevious = kineticEnergy;
  kineticEnergy = 0;

  //calc nodes
  for (int i = 1; i < nodes.Length; i++)
  {
    Node n = nodes[i];
    n.Calculate(timeIncrement, lambda);
    //n.calcPosition(); //remove
```

```
  kineticEnergy += n.CalcKineticEnergy();
}

//damp if required
kE3 = kineticEnergy;

if (kineticDamping) //if kinetic damping enabled
{
  double beta = 0;

  // there is a peak between kE1 and kE3
  // then damp system at this point
  if (kE2 > kE1 && kE3 < kE2)
  {
    //check when last damped
    if (this.iterations - lastDampedIterationNum <=
        MIN_ITERATIONS_BEFORE_DAMPING)
    {
      //too soon, so skip damping
    }
    else
    {
      //DAMP NOW:
      lastDampedIterationNum = this.iterations;

      peakCount++;
      beta = (kE3 - kE2) / (kE3 - 2 * kE2 + kE1);

      /*
      if (peakCount % lamdaUseCount == 0)
      {
        //lambda *= lambdaIncreaseFactor;
        //lambdaFactor *= lambdaIncreaseFactor;
      }
      */

      for (int i = 1; i < nodes.Length; i++)
```

```
        {
          Node n = nodes[i];
          n.CalcDisplacementAtKineticEnergyPeak(beta);
          n.Velocity = new Vector();
          n.ForceResidual = new Vector();
          //kineticEnergy += n.CalcKineticEnergy();
          kineticEnergyMax = kineticEnergy;
        }
      }
    }
  }
}

public void DrawNodes(bool drawForces)
{
  //dwg.ClearViewport();

  // draw nodes
  for ( int i = 1; i < nodes.Length; i++ )
  {
    Node n = nodes[i];
    Point3D centrePoint = new Point3D(n.X, n.Y, n.Z);

    Brush brush;

    if ( n.Fixity.Magnitude() > 0 )
    {
      //restrained in some direction
      brush = Brushes.Red;
    }
    else
    {
      if (n.ForceApplied.Magnitude() > 0)
      {
        brush = Brushes.DarkTurquoise;
      }
      else
```

```
      {
        brush = Brushes.Green;
      }
    }

    // if node selected in treeview
    if (n.nodeNumber == selectedNodeNum)
    {
      brush = Brushes.Orange;
    }

    double nodeRadius;
    //nodeRadius = 0.07
    nodeRadius = memberTypes[2].radius * 1.5;

    drawing.DrawSphere(centrePoint, nodeRadius, brush);

    // draw the force applied to the node
    if (drawForces)
    {
      //if (n.ForceApplied.Magnitude() != 0)
      //{
        DrawForce(n);
      //}
    }
  }
}

public int NodeCount()
{
  return iNodeCount;
}

public int MemberCount()
{
  return iMemberCount;
}
```

```
public void SetDrawingControl(_3D_Drawing_Control.
    UserControl1 dwg)
{
  drawing = dwg;
}

public void Draw(bool drawMembers, bool drawNodes, bool
    drawForces)
{
  if (drawMembers) { DrawMembers(); }
  if (drawNodes) { DrawNodes(drawForces); }
}

public void DrawMembers()
{
  //drawing.ClearViewport();


  // draw members
  for (int i = 1; i < members.Length; i++)
  {
    Member m = members[i];
    Point3D p1 = new Point3D(m.StartNode.X, m.StartNode.Y
        , m.StartNode.Z);
    Point3D p2 = new Point3D(m.EndNode.X, m.EndNode.Y, m.
        EndNode.Z);
    //dwg.DrawLine(p1, p2);
    //dwg.DrawBox(p1, p2);

    //add displacements
    p1.Offset(m.StartNode.Displacement.X, m.StartNode.
        Displacement.Y, m.StartNode.Displacement.Z);
    p2.Offset(m.EndNode.Displacement.X, m.EndNode.
        Displacement.Y, m.EndNode.Displacement.Z);

    double memberRadius;
```

```
Brush brush;

if (m.memberType.typeNumber == 1)
{
  //cable
  //memberRadius = 0.015;
  //memberRadius = 0.01;
  memberRadius = m.memberType.radius*3;
  //brush = Brushes.Black;
  brush = Brushes.DarkGreen;
}
else
{
  //strut
  //memberRadius = 0.05;
  memberRadius = m.memberType.radius;
  brush = Brushes.SteelBlue;
}

// if member selected in treeview
if (m.memberNumber == selectedMemberNum)
{
  brush = Brushes.Orange;
}

// if memberforce is zero then draw different color
if (m.MemberForce.Magnitude() == 0)
{
  //brush = Brushes.DarkGray;
}

drawing.DrawCylinder(p1, p2, memberRadius, brush);

/*
// Member Force Arrows
double lenChange = m.ChangeInLength();
```

```
if (lenChange < 0)
{
  //compression
  brush = Brushes.SteelBlue;
}
else if(lenChange > 0)
{
  //tension
  brush = Brushes.Yellow;
}
else
{
  //no force
  brush = Brushes.DarkGray;
}

//draw member force
Node n = new Node();
n.X = p1.X;
n.Y = p1.Y;
n.Z = p1.Z;
DrawForceArrow(m.ForceOnStartNode, n, brush);

//on node
DrawForceArrow(m.ForceOnStartNode, m.StartNode, brush
    );

n = new Node();
n.X = p2.X;
n.Y = p2.Y;
n.Z = p2.Z;
DrawForceArrow(m.ForceOnEndNode, n, brush);

//on node
DrawForceArrow(m.ForceOnEndNode, m.EndNode, brush);
*/
```

```
  }
}

private void DrawForce(Node n)
{
  DrawForceArrow(n.ForceApplied.multiplyByValue(1), n,
     Brushes.SpringGreen);
  //DrawForceArrow(n.ForceInternal.multiplyByValue(1), n,
      Brushes.Orange);
  DrawForceArrow(n.ForceResidual.multiplyByValue(1), n,
     Brushes.Plum);
}

private void DrawForceArrow(Vector ForceDirection, Node n
   , Brush brush)
{
  //Vector ForceDirection = n.ForceApplied.
     multiplyByValue(1); // copy vector
  ForceDirection.MakeUnitVector();
  ForceDirection = ForceDirection.multiplyByValue(-1);

  double memberRadius = memberTypes[2].radius;

  double forceDisplayOffset = memberRadius * 6; //0.03;
     // distance of tip of arrow from node
  double forceDisplayLength = memberRadius * 16; //0.08;
     // length of arrow

  Vector p1Loc = ForceDirection.multiplyByValue(
     forceDisplayOffset);
  p1Loc.addVectorToThis(n);

  Vector p2Loc = ForceDirection.multiplyByValue(
     forceDisplayLength + forceDisplayOffset);
  p2Loc.addVectorToThis(n);

  Point3D p1 = new Point3D(p1Loc.X, p1Loc.Y, p1Loc.Z);
```

```
  Point3D p2 = new Point3D(p2Loc.X, p2Loc.Y, p2Loc.Z);

  double cylRadius = memberRadius / 2;
  double sphRadius = memberRadius;

  drawing.DrawCylinder(p1, p2, cylRadius, brush);
  drawing.DrawSphere(p1, sphRadius, brush);
}

//save class to file
public void Write(string fileName)
{
  //Open a FileStream on the file "aboutme"
  FileStream fout = new FileStream(fileName, FileMode.
     OpenOrCreate,
  FileAccess.Write, FileShare.ReadWrite);

  //Create a BinaryWriter from the FileStream
  BinaryWriter bw = new BinaryWriter(fout);

  int nodeCount = nodes.Length;

  bw.Write(nodeCount);

  for (int i = 1; i < nodeCount; i++)
  {
    Node n = nodes[i];

    // co-ordinates
    bw.Write(n.X);
    bw.Write(n.Y);
    bw.Write(n.Z);

    // Fixity
    bw.Write(n.Fixity.X);
    bw.Write(n.Fixity.Y);
    bw.Write(n.Fixity.Z);
```

```
  // ForceApplied
  bw.Write(n.ForceApplied.X);
  bw.Write(n.ForceApplied.Y);
  bw.Write(n.ForceApplied.Z);

  // Displacement
  bw.Write(n.Displacement.X);
  bw.Write(n.Displacement.Y);
  bw.Write(n.Displacement.Z);

}

int memberCount = members.Length;

bw.Write(memberCount);

for (int i = 1; i < memberCount; i++)
{
  Member m = members[i];

  // Node Numbers
  bw.Write(m.StartNode.nodeNumber);
  bw.Write(m.EndNode.nodeNumber);

  // Member Type
  bw.Write(m.memberType.typeNumber);

  // StiffnessForce
  bw.Write(m.StiffnessForce.X);
  bw.Write(m.StiffnessForce.Y);
  bw.Write(m.StiffnessForce.Z);

  // TensionForce
  bw.Write(m.TensionForce.X);
  bw.Write(m.TensionForce.Y);
  bw.Write(m.TensionForce.Z);
```

```
  }

  //Close the file and free resources
  bw.Close();
}

//save for abaqus input
public void SaveForAbaqus(string fileName)
{
  //Open a FileStream
  System.IO.StreamWriter objWriter = new System.IO.
     StreamWriter(fileName, false);

  objWriter.WriteLine("*Node");

  int nodeCount = nodes.Length;

  string topNodesSet = "";

  for (int i = 1; i < nodeCount; i++)
  {
    Node n = nodes[i];

    objWriter.WriteLine(i + ", " + n.X + ", " + n.Y + ",
       " + n.Z);

    if (n.nodeLabel == "top")
    {
      topNodesSet += ", " + i;
    }
  }

  int memberCount = members.Length;

  objWriter.WriteLine("*Element, type=T3D2");

  string cableSet = "";
```

```
string strutSet = "";

for (int i = 1; i < memberCount; i++)
{
  Member m = members[i];

  // Node Numbers
  objWriter.WriteLine(i + ", " + m.StartNode.nodeNumber
      + ", " + m.EndNode.nodeNumber);

  if (m.memberType.name == "cable")
  {
    cableSet += ", " + i;
  }
  else
  {
    strutSet += ", " + i;
  }
}

//cable set
//*Elset, elset=_PickedSet8, internal
//2,  3,  4,  5,  6,  7,  8,  9, 13, 14, 15, 16
objWriter.WriteLine("*Elset, elset=CableSet, internal")
   ;
cableSet = cableSet.Trim(',');
cableSet = cableSet.Trim();
objWriter.WriteLine(cableSet);

//strut set
objWriter.WriteLine("*Elset, elset=StrutSet, internal")
   ;
strutSet = strutSet.Trim(',');
strutSet = strutSet.Trim();
objWriter.WriteLine(strutSet);

//top nodes set
```

```
  objWriter.WriteLine("*Elset, elset=TopNodesSet,
     internal");
  topNodesSet = topNodesSet.Trim(',');
  topNodesSet = topNodesSet.Trim();
  objWriter.WriteLine(topNodesSet);

  //section assignments
  objWriter.WriteLine("** Section: StrutSection");
  objWriter.WriteLine("*Solid Section, elset=StrutSet,
     material=Aluminium");
  objWriter.WriteLine("3.55314e-05,");
  objWriter.WriteLine("** Section: CableSection");
  objWriter.WriteLine("*Solid Section, elset=CableSet,
     material=\"Stainless Steel\"");
  objWriter.WriteLine("7.85398e-07,");

  //Close the file and free resources
  objWriter.Close();
}

//read class from file
public void Read(string fileName)
{
  //Open a FileStream
  FileStream fin = new FileStream(fileName, FileMode.Open
     ,
  FileAccess.Read, FileShare.ReadWrite);

  //Create a BinaryReader from the FileStream
  BinaryReader br = new BinaryReader(fin);

  //Seek to the start of the file
  br.BaseStream.Seek(0, SeekOrigin.Begin);

  //Read from the file and store the values to the
     variables
  int nodeCount = br.ReadInt32();
```

```
for (int i = 1; i < nodeCount; i++)
{
  // co-ordinates
  double nX = br.ReadDouble();
  double nY = br.ReadDouble();
  double nZ = br.ReadDouble();

  // Fixity
  double fX = br.ReadDouble();
  double fY = br.ReadDouble();
  double fZ = br.ReadDouble();

  Node n = addNode(new Point3D(nX, nY, nZ), new Vector(
     fX, fY, fZ));

  // ForceApplied
  n.ForceApplied.X = br.ReadDouble();
  n.ForceApplied.Y = br.ReadDouble();
  n.ForceApplied.Z = br.ReadDouble();

  // Displacement
  n.Displacement.X = br.ReadDouble();
  n.Displacement.Y = br.ReadDouble();
  n.Displacement.Z = br.ReadDouble();

}

int memberCount = br.ReadInt32();

for (int i = 1; i < memberCount; i++)
{
  // Node Numbers
  int startNodeNum = br.ReadInt32();
  int endNodeNum = br.ReadInt32();

  // Member Type
```

```
    int memberTypeNum = br.ReadInt32();

    Member m = addMember(startNodeNum, endNodeNum,
       memberTypeNum);

    // StiffnessForce
    m.StiffnessForce.X = br.ReadDouble();
    m.StiffnessForce.Y = br.ReadDouble();
    m.StiffnessForce.Z = br.ReadDouble();

    // TensionForce
    m.TensionForce.X = br.ReadDouble();
    m.TensionForce.Y = br.ReadDouble();
    m.TensionForce.Z = br.ReadDouble();
  }

  //Close the file and free resources
  br.Close();
}

//add displacements to node positions and set that as new
    position
//reset velocites etc.
//set member force as tension (pretension)
public void SetCurrentConfigAsInitialConfig()
{
  for (int i = 1; i < nodes.Length; i++)
  {
    nodes[i].SetNewPosition();
    nodes[i].Velocity = new Vector();
    nodes[i].VelocityPrevious = new Vector();
    nodes[i].Displacement = new Vector();
    nodes[i].ForceInternal = new Vector();
    nodes[i].ForceResidual = new Vector();
  }

  for (int i = 1; i < members.Length; i++)
```

```
      {
        Vector PreTension = members[i].MemberForce.
            multiplyByValue(1);
        //Node SNode = members[i].StartNode;
        //Node ENode = members[i].EndNode;

        //members[i] = new Member(SNode, ENode);
        members[i].TensionForce = PreTension;
        members[i].CalcOriginalLength();
      }
    }

    //reset to the initial config
    public void ResetConfig()
    {
      for (int i = 1; i < nodes.Length; i++)
      {
        nodes[i].Velocity = new Vector();
        nodes[i].VelocityPrevious = new Vector();
        nodes[i].Displacement = new Vector();
        nodes[i].ForceInternal = new Vector();
        nodes[i].ForceResidual = new Vector();
      }
    }
  }
}
```

A.6 ModelBuilding Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using System.Windows.Media.Media3D;
//using System.Windows.Forms.DataVisualization.Charting;

namespace Tensegrity_01
{
  public partial class Form1 : Form
  {
    void BuildModel(int modelIndex)
    {
      switch (modelIndex)
      {
        case 0:
          T1.name = "Strut";
          BuildStrut();
          break;

        case 1:
          BuildCableCantilever1();
          break;

        case 2:
          BuildCableCantilever2();
          break;

        case 3:
```

```
  BuildCableCantilever3();
  break;

case 4:
  BuildSnelsonX();
  break;

case 5:
  BuildPrism();
  break;

case 6:
  BuildSquarePrism(new Point3D(0, 0, 0));
  break;

case 7:
  BuildBeam();
  break;

case 8:
  BuildMechanism();
  break;

case 9:
  BuildMechanism2();
  break;

case 10:
  BuildStrut2();
  break;

case 11:
  T1.name = "Square Tower";
  BuildSquareTower();
  break;

case 12: //Tower
```

```
  T1.name = "Tower";

  //Standard Tower (physical model)
  BuildTower(6, 3, 0.340, 0.15);
  T1.activeNode = 31;
  T1.setTotalAppliedVerticalForce(-600);

  //Parametric Tower
  //int layerCount = 5;
  //double layerHeight = 0.340 * 3 / layerCount;
  //BuildTower(6, layerCount, layerHeight, 0.15);

  //T1.setNodeAppliedVerticalForce("top", -100);
  break;

case 13: //4 Strut module
  T1.name = "4 Strut";

  //Standard Module (physical model)
  BuildTower(4, 1, 0.260, 0.15);
  T1.activeNode = 8;

  //T1.setMemberLength("vertical", 0.280);
  //T1.setMemberLength("top", 0.190);
  //T1.setMemberLength("base", 0.190);
  //T1.setNodeAppliedVerticalForce("top", -150);
  //T1.setNodeDisplacement("top", -0.030);

  T1.setTotalAppliedVerticalForce(-560); //model
  //T1.setTotalAppliedVerticalForce(400); //tension

  //T1.setTotalAppliedVerticalForce(-800); //ABAQUS
  break;

case 14: //6 Strut module
  BuildTower(6, 1, 0.260, 0.15);
  T1.activeNode = 12;
```

```
  T1.setNodeAppliedVerticalForce("top", -150);
  break;

case 15: // Motro Simplex
  BuildTower(3, 1, 0.260, 0.15);
  T1.activeNode = 6;
  break;

case 16: //Variable Modules
  int strutCount = 6;
  BuildTower(strutCount, 1, 0.260, 0.15);
  //T1.activeNode = strutCount * 2;

  /*
  T1.name = "4 strut squashing 2 nodes";
  T1.nodes[6].ForceApplied.X = 100;
  T1.nodes[6].ForceApplied.Z = 100;
  T1.nodes[8].ForceApplied.X = -100;
  T1.nodes[8].ForceApplied.Z = -100;
  T1.activeNode = 6;
  */

  T1.name = "4 strut compression - variable cable
     tension";
  //T1.setTotalAppliedVerticalForce(-560);
  T1.SetCableLengthFactor(1.001);
  T1.SetCableLengthFactor(0.98, "vertical");
  //T1.setNodeDisplacement("top", -0.001);
  T1.activeNode = 8;

  /*
  T1.name = "4 strut squashing 4 nodes";
  T1.nodes[6].ForceApplied.X = 100;
  T1.nodes[6].ForceApplied.Z = 100;
  T1.nodes[8].ForceApplied.X = -100;
  T1.nodes[8].ForceApplied.Z = -100;
  T1.nodes[5].ForceApplied.X = -100;
```

```
      T1.nodes[5].ForceApplied.Z = 100;
      T1.nodes[7].ForceApplied.X = 100;
      T1.nodes[7].ForceApplied.Z = -100;
      T1.activeNode = 6;
      */

      /*
      T1.name = "4 strut lateral 2 nodes";
      T1.nodes[7].ForceApplied.Z = -100;
      T1.nodes[8].ForceApplied.Z = -100;
      T1.activeNode = 7;
      */

      break;

    case 17: //Variable Modules
      BuildBracedTower();
      break;

    case 18: //Variable Modules
      BuildTruss1();
      break;

    case 19: //Variable Modules
      BuildTrussTwoMember();
      break;
  }
}

void BuildStrut()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes for snelson X
```

```
  T1.addNode(new Point3D(0, 0, 0), new Vector(0, 1, 0));
  T1.addNode(new Point3D(0, 0.3, 0));
  T1.activeNode = 2;

  //forces
  T1.nodes[2].ForceApplied.Y = -2e6; //down on top node
  //T1.nodes[1].ForceApplied.Y = 10; //up on bottom node

  //define member info
  T1.addMember(1, 2, 2);
}

void BuildStrut2()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(1, 1, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes for snelson X
  double width = 0.1;
  T1.addNode(new Point3D(0, 0, 0), new Vector(1, 1, 0));
  T1.addNode(new Point3D(width / 2, 0.3, 0));
  T1.addNode(new Point3D(width, 0, 0), new Vector(1, 1,
     0));

  //forces
  T1.nodes[2].ForceApplied.Y = -2e6; //down on top node
  //T1.nodes[1].ForceApplied.Y = 10; //up on bottom node

  T1.activeNode = 2;

  //define member info
  T1.addMember(1, 2, 2);
  T1.addMember(2, 3, 2);
}
```

```
//2 Cable
void BuildMechanism()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0, 1, 2));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes
  T1.addNode(new Point3D(-0.3, 1, 0), new Vector(1, 1, 0)
     );
  T1.addNode(new Point3D(0, 1, 0), new Vector(0, 0, 0));
  T1.addNode(new Point3D(0.3, 1, 0), new Vector(1, 1, 0))
     ;

  //forces
  T1.nodes[2].ForceApplied.Y = -1000; //down on middle
     node

  //define member info
  T1.addMember(1, 2, 1);
  T1.addMember(2, 3, 1);

  T1.activeNode = 2;
}

//4 Cable Mechanism
void BuildMechanism2()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0, 1, 2));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes
  T1.addNode(new Point3D(-0.3, 1, 0), new Vector(1, 1, 0)
     );
```

```
  T1.addNode(new Point3D(-0.15, 1, 0), new Vector(0, 0,
     0));
  T1.addNode(new Point3D(0, 1, 0), new Vector(0, 0, 0));
  T1.addNode(new Point3D(0.15, 1, 0), new Vector(0, 0, 0)
     );
  T1.addNode(new Point3D(0.3, 1, 0), new Vector(1, 1, 0))
     ;

  //forces
  T1.activeNode = 3;
  T1.nodes[3].ForceApplied.Y = -1000; //down on middle
     node

  //define member info
  T1.addMember(1, 2, 1);
  T1.addMember(2, 3, 1);
  T1.addMember(3, 4, 1);
  T1.addMember(4, 5, 1);
}

void BuildCableCantilever1()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes for snelson X
  T1.addNode(new Point3D(0, 0, 0), new Vector(1, 1, 0));
  T1.addNode(new Point3D(0.3, 0, 0));
  T1.addNode(new Point3D(0, 0.3, 0), new Vector(1, 1, 0))
     ;

  //forces
  T1.nodes[2].ForceApplied.Y = -500;

  //define member info
```

```
  T1.addMember(1, 2, 2); //.TensionForce.Y = 84;
  T1.addMember(2, 3, 1); //.TensionForce.Y = -11.5;

  T1.activeNode = 2;

  //T1.members[1].SetTension(10000);
}

void BuildCableCantilever2()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes for snelson X
  T1.addNode(new Point3D(0, 0, 0), new Vector(1, 1, 0));
  T1.addNode(new Point3D(0.3, 0.3, 0));
  T1.addNode(new Point3D(0, 0.3, 0), new Vector(1, 1, 0))
     ;

  //forces
  T1.nodes[2].ForceApplied.Y = -10000;

  //define member info
  T1.addMember(1, 2, 2);
  T1.addMember(2, 3, 1);

  T1.activeNode = 2;
}

void BuildCableCantilever3()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));
```

```
  //define coordinates of nodes for snelson X
  T1.addNode(new Point3D(1, 0, 0), new Vector(1, 1, 0));
  T1.addNode(new Point3D(2, 0.5, 0));
  T1.addNode(new Point3D(1, 1, 0), new Vector(1, 1, 0));

  //forces
  T1.nodes[2].ForceApplied.X = -10;

  //define member info
  T1.addMember(1, 2, 1);
  T1.addMember(2, 3, 1);
  T1.addMember(1, 3, 2);

  T1.activeNode = 2;
}

//Truss
void BuildTruss1()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(3, 3, 25));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //member sections
  T1.Strut.Area = 1200E-6; //mm2 = 10E-6 m2
  T1.Strut.E = 200E9; //200 GPa

  //nodes
  T1.addNode(new Point3D(0, 0, 0), new Vector(1, 1, 1));
     //A
  T1.addNode(new Point3D(4, 0, 0), new Vector(0, 0, 1));
     //B
  T1.addNode(new Point3D(7, 0, 0), new Vector(0, 1, 1));
     //C
  T1.addNode(new Point3D(4, 4, 0), new Vector(0, 0, 1));
```

```
      //D

  //forces
  T1.activeNode = 2;
  T1.nodes[4].ForceApplied.X = -35000;
  T1.nodes[2].ForceApplied.Y = -84000;

  //define member info
  T1.addMember(1, 2, 2);
  T1.addMember(2, 3, 2);
  T1.addMember(1, 4, 2);
  T1.addMember(4, 2, 2);
  T1.addMember(4, 3, 2);
}

void BuildTrussTwoMember()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
      -1));

  //member sections
  //T1.Strut.radius = 0.3; //for drawing purposes
  T1.Strut.Area = 0.009677; //96.77cm2
  T1.Strut.E = 68940749500; //703000 kg/cm2

  //nodes
  double w = 0.6599; //65.99cm;
  double h = 0.1905; //19.05cm;
  T1.addNode(new Point3D(0, 0, 0), new Vector(1, 1, 1));
      //A
  T1.addNode(new Point3D(w / 2, h, 0), new Vector(0, 0,
      1)); //B
  T1.addNode(new Point3D(w, 0, 0), new Vector(1, 1, 1));
      //C
```

```
  //forces
  T1.activeNode = 2;
  T1.nodes[2].ForceApplied.Y = -500;

  //define member info
  T1.addMember(1, 2, 2);
  T1.addMember(2, 3, 2);
}

void BuildBracedTower()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes
  double h = 0.3;
  double w = 0.1;

  //base
  T1.addNode(new Point3D(-2 * w, 0, 0), new Vector(1, 1,
     1)); // no displacement in any direction
  T1.addNode(new Point3D(2 * w, 0, 0), new Vector(1, 1,
     1));

  //level 1
  T1.addNode(new Point3D(-w, h, 0));
  T1.addNode(new Point3D(w, h, 0));

  //top
  T1.addNode(new Point3D(0, 2 * h, 0));

  //forces
  T1.activeNode = 5;
  T1.nodes[5].ForceApplied.Y = -50000;
```

```
  //define member info
  T1.addMember(1, 3, 2);
  T1.addMember(3, 5, 2);
  T1.addMember(5, 4, 2);
  T1.addMember(4, 2, 2);
  T1.addMember(2, 3, 2);
  T1.addMember(1, 4, 2);
  T1.addMember(3, 4, 2);
}

void BuildSnelsonX()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
     -1));

  //define coordinates of nodes for snelson X
  double sideLen = 0.3;
  T1.addNode(new Point3D(0, 0, 0), new Vector(1, 1, 1));
     // no displacement in any direction
  T1.addNode(new Point3D(0, sideLen, 0));
  T1.addNode(new Point3D(sideLen, sideLen, 0));
  T1.addNode(new Point3D(sideLen, 0, 0), new Vector(1, 1,
      0)); // displ. only restrained in y (chged)

  //forces
  //T1.nodes[2].ForceApplied.Y = -5;
  T1.nodes[3].ForceApplied.Y = -500;
  //T1.nodes[3].ForceApplied.X = 5;

  //T1.nodes[2].ForceApplied.X = 5;
  //T1.nodes[4].ForceApplied.X = -5;

  //T1.nodes[3].ForceApplied.X = 10;

  //define member info
```

```
  T1.addMember(1, 2, 1);
  T1.addMember(2, 3, 1);
  T1.addMember(3, 4, 1);
  T1.addMember(1, 4, 1);
  T1.addMember(1, 3, 2);
  T1.addMember(2, 4, 2);
}

//Motro's Simplex
//basic tensegrity prism (3strut -> "Simplex")
void BuildPrism()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(5, 6, 5));
  dwgControl.SetCameraLookDirection(new Vector3D(-1, -1,
     -1));

  double ratio = 1.468;
  double strutLength = 0.3;
  double cableLength = strutLength / ratio;
  double radius = cableLength * Math.Sin(Math.PI / 6) /
     Math.Sin(Math.PI / 3 * 2);
  double w = strutLength * strutLength / 2 / cableLength;
  //w = strutLength * ratio / 2; //approx. the same
  double zz = Math.Acos(ratio / 2);
  double layerHeight = w * Math.Tan(Math.Acos(ratio / 2))
     ;

  double twistAngle = Math.PI / 3 * 2 - Math.PI / 6;
  BuildTower(3, 1, layerHeight, radius, twistAngle);

  T1.activeNode = 6;
  T1.nodes[6].ForceApplied.Y = -1000;
}

void BuildBeam()
{
```

```
//set camera
dwgControl.SetCameraPosition(new Point3D(0.5, 0.5, 10))
    ;
dwgControl.SetCameraLookDirection(new Vector3D(0, 0,
    -1));

int beamWidth = 5; //half width

for (int a = -beamWidth; a < beamWidth; a++)
{
  T1.addNode(new Point3D(a + 0.5, 2, 0));
  T1.addNode(new Point3D(a + 0.5, 0, 0));
}

int nodeCount = T1.NodeCount();

//struts
for (int a = 1; a < nodeCount - 5; a += 4)
{
  T1.addMember(a, a + 5, 2);
  T1.addMember(a + 3, a + 6, 2);
}

//extra end members
T1.addMember(2, 3, 2);
T1.addMember(nodeCount - 3, nodeCount, 2);

//cables
for (int a = 1; a <= nodeCount - 1; a += 2)
{
  T1.addMember(a, a + 1, 1);


  if (a <= nodeCount - 3)
  {
    T1.addMember(a, a + 2, 1);
  }
```

```
    if (a <= nodeCount - 3)
    {
      T1.addMember(a, a + 3, 1);
      T1.addMember(a + 1, a + 3, 1);
    }

    if (a <= nodeCount - 4)
    {
      T1.addMember(a + 3, a + 4, 1);
    }
  }

  T1.nodes[1].Fixity = new Vector(0, 1, 0);
  T1.nodes[2].Fixity = new Vector(0, 1, 0);
  //T1.nodes[nodeCount-1].fixity = 1;
  //T1.nodes[nodeCount].fixity = 1;

  //T1.DrawNodes(dwgControl);

  //forces
  T1.nodes[20].ForceApplied.Y = -10;

}

void BuildSquarePrism(Point3D cornerPoint)
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(4, 4, 4));
  dwgControl.SetCameraLookDirection(new Vector3D(-1, -1,
     -1));

  double width = 1;
  double height = 1;

  double edgeNodes = 3;
```

```
/*
for (int a = 0; a < edgeNodes; a++)
{
  double xPos = a / (edgeNodes - 1) * width;

  for (int b = 0; b < edgeNodes; b++)
  {
    double zPos = b / (edgeNodes - 1) * width;

    for (int c = 0; c < 2; c++)
    {
      T1.addNode(new Point3D(xPos, c * height, zPos));
    }
  }
}
*/

double yPos;
yPos = cornerPoint.Y;

//base corners
Node p1s = T1.addNode(cornerPoint.X, yPos, cornerPoint.
   Z);
Node p4s = T1.addNode(cornerPoint.X + width, yPos,
   cornerPoint.Z);
Node p2s = T1.addNode(cornerPoint.X, yPos, cornerPoint.
   Z + width);
Node p3s = T1.addNode(cornerPoint.X + width, yPos,
   cornerPoint.Z + width);

yPos = cornerPoint.Y + height;

//midpoints
Node p3e = T1.addNode(cornerPoint.X + width / 2, yPos,
   cornerPoint.Z);
Node p4e = T1.addNode(cornerPoint.X, yPos, cornerPoint.
   Z + width / 2);
```

```
Node p2e = T1.addNode(cornerPoint.X + width, yPos,
    cornerPoint.Z + width / 2);
Node p1e = T1.addNode(cornerPoint.X + width / 2, yPos,
    cornerPoint.Z + width);

// Members
// struts
T1.addMember(p1s, p1e, 2);
T1.addMember(p2s, p2e, 2);
T1.addMember(p3s, p3e, 2);
T1.addMember(p4s, p4e, 2);

//cables
T1.addMember(p2s, p1e, 1);
T1.addMember(p3s, p2e, 1);
T1.addMember(p4s, p3e, 1);
T1.addMember(p1s, p4e, 1);

T1.addMember(p1s, p2s, 1);
T1.addMember(p2s, p3s, 1);
T1.addMember(p3s, p4s, 1);
T1.addMember(p4s, p1s, 1);

T1.addMember(p1e, p2e, 1);
T1.addMember(p2e, p3e, 1);
T1.addMember(p3e, p4e, 1);
T1.addMember(p4e, p1e, 1);

T1.addMember(p1s, p3e, 1);
T1.addMember(p2s, p4e, 1);
T1.addMember(p3s, p1e, 1);
T1.addMember(p4s, p2e, 1);

//top
//T1.addMember(p1e, p3e, 1);
//T1.addMember(p2e, p4e, 1);
//T1.addMember(p3e, p1e, 1);
```

```
//T1.addMember(p4e, p2e, 1);

// boundary conditions
p1s.Fixity = new Vector(1, 1, 1);
p2s.Fixity = new Vector(0, 1, 0);
p3s.Fixity = new Vector(1, 1, 0);
p4s.Fixity = new Vector(0, 1, 0);

// loading
//p1e.ForceApplied.X = 10;
p1e.ForceApplied.X = -10;
p2e.ForceApplied.Y = -10;
p3e.ForceApplied.Y = -10;
p4e.ForceApplied.Y = -10;

//pre-stress
/*
T1.members[1].TensionForce.Y = -4.6;
T1.members[2].TensionForce.Y = -3.5;
T1.members[3].TensionForce.Y = -4.6;
T1.members[4].TensionForce.Y = -3.5;
T1.members[5].TensionForce.Y = -4.9;
T1.members[6].TensionForce.Y = -3.5;
T1.members[7].TensionForce.Y = -4.9;
T1.members[8].TensionForce.Y = -3.5;

T1.members[13].TensionForce.Y = 126.3;
T1.members[14].TensionForce.Y = 126.3;
T1.members[15].TensionForce.Y = 126.3;
T1.members[16].TensionForce.Y = 126.3;
T1.members[17].TensionForce.Y = -4.9;
T1.members[18].TensionForce.Y = -3.5;
T1.members[19].TensionForce.Y = -4.8;
T1.members[20].TensionForce.Y = -3.5;
*/

/*
```

```
  for (int i = 1; i < T1.members.Length; i++)
  {
    Member m = T1.members[i];

    if (m.memberType.typeNumber == 1) // cable
    {
      m.CalcNodeDirections();
      m.SetTension(-1000);
    }
  }
  */
}

void BuildSquareTower()
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(8, 9, 8));
  dwgControl.SetCameraLookDirection(new Vector3D(-1,
     -0.9, -1));

  double layers = 4;

  Node Top1 = new Node();
  Node Top2 = new Node();
  Node Top3 = new Node();
  Node Top4 = new Node();

  for (int a = 1; a <= layers; a++)
  {
    double yBottom = a - 1;
    double yTop = a;

    //struts
    Node N1s = T1.GetNode(0, yBottom, 0);
    Node N1e = T1.GetNode(0.5, yTop, 1);
    T1.addMember(N1s, N1e, 2);
```

```
Node N2s = T1.GetNode(0, yBottom, 1);
Node N2e = T1.GetNode(1, yTop, 0.5);
T1.addMember(N2s, N2e, 2);

Node N3s = T1.GetNode(1, yBottom, 1);
Node N3e = T1.GetNode(0.5, yTop, 0);
T1.addMember(N3s, N3e, 2);

Node N4s = T1.GetNode(1, yBottom, 0);
Node N4e = T1.GetNode(0, yTop, 0.5);
T1.addMember(N4s, N4e, 2);

//cables
T1.addMember(N1e, N2s, 1);
T1.addMember(N2e, N3s, 1);
T1.addMember(N3e, N4s, 1);
T1.addMember(N4e, N1s, 1);

//T1.addMember(N1e, N3s, 1);
//T1.addMember(N2e, N4s, 1);
//T1.addMember(N3e, N1s, 1);
//T1.addMember(N4e, N2s, 1);

//top cables
double yTopLevel = yTop;
T1.addMember(T1.GetNode(0, yTopLevel, 0.5), T1.
    GetNode(0.5, yTopLevel, 1), 1);
T1.addMember(T1.GetNode(0.5, yTopLevel, 1), T1.
    GetNode(1, yTopLevel, 0.5), 1);
T1.addMember(T1.GetNode(1, yTopLevel, 0.5), T1.
    GetNode(0.5, yTopLevel, 0), 1);
T1.addMember(T1.GetNode(0.5, yTopLevel, 0), T1.
    GetNode(0, yTopLevel, 0.5), 1);

if (a == layers)
{
  Top1 = N1e;
```

```
    Top2 = N2e;
    Top3 = N3e;
    Top4 = N4e;
  }
}

//connection between modules (layers)
for (int a = 1; a <= layers - 1; a++)
{
  double yCon = a;
  T1.addMember(T1.GetNode(0, yCon, 0), T1.GetNode(0,
      yCon, 0.5), 1);
  T1.addMember(T1.GetNode(0, yCon, 0.5), T1.GetNode(0,
      yCon, 1), 1);
  T1.addMember(T1.GetNode(0, yCon, 1), T1.GetNode(0.5,
      yCon, 1), 1);
  T1.addMember(T1.GetNode(0.5, yCon, 1), T1.GetNode(1,
      yCon, 1), 1);

  T1.addMember(T1.GetNode(1, yCon, 1), T1.GetNode(1,
      yCon, 0.5), 1);
  T1.addMember(T1.GetNode(1, yCon, 0.5), T1.GetNode(1,
      yCon, 0), 1);
  T1.addMember(T1.GetNode(1, yCon, 0), T1.GetNode(0.5,
      yCon, 0), 1);
  T1.addMember(T1.GetNode(0.5, yCon, 0), T1.GetNode(0,
      yCon, 0), 1);
}

//base cables
Node Bot1 = T1.GetNode(0, 0, 0);
Node Bot2 = T1.GetNode(0, 0, 1);
Node Bot3 = T1.GetNode(1, 0, 1);
Node Bot4 = T1.GetNode(1, 0, 0);

T1.addMember(Bot1, Bot2, 1);
T1.addMember(Bot2, Bot3, 1);
```

```
  T1.addMember(Bot3, Bot4, 1);
  T1.addMember(Bot4, Bot1, 1);

  //BC
  Bot1.Fixity = new Vector(1, 1, 1);
  Bot2.Fixity = new Vector(1, 1, 1);
  Bot3.Fixity = new Vector(1, 1, 1);
  Bot4.Fixity = new Vector(1, 1, 1);

  //loads
  Top1.ForceApplied.Y = -10;
  Top2.ForceApplied.Y = -10;
  Top3.ForceApplied.X = -10;
  Top4.ForceApplied.X = -10;
}

// overload
void BuildTower(int strutCount, int layerCount, double
    layerHeight, double radius)
{
  double angle = Math.PI * 2 / strutCount; //radians
  double twistAngle = angle / 2;
  BuildTower(strutCount, layerCount, layerHeight, radius,
      twistAngle);
}

void BuildTower(int strutCount, int layerCount, double
    layerHeight, double radius, double twistAngle)
{
  //set camera
  //dwgControl.SetCameraPosition(new Point3D(1, 1.15, 1))
     ;
  //dwgControl.SetCameraPosition(new Point3D(1, 1.5, 1));
      //Tower
  dwgControl.SetCameraPosition(new Point3D(1, 1.5, 1));
     //Module
  dwgControl.SetCameraLookDirection(new Vector3D(-1, -1,
```

```
   -1));

//int strutCount = 4;
//int layerCount = 2;
//double layerHeight = 0.260;
//double radius = 0.150;

double angle = Math.PI * 2 / strutCount; //radians

Node[,] TowerNodes = new Node[layerCount * 2 + 1,
   strutCount];

int levelNum = 0;

for (int c = 1; c <= layerCount; c++)
{
  for (int b = 1; b <= 2; b++)
  {
    for (int a = 1; a <= strutCount; a++)
    {
      double bearingAngle = angle * a;

      //rotate top layer of nodes (every odd layer)
      if (b == 2)
      {
        //bearingAngle += angle / 2;
       bearingAngle += twistAngle;
      }

      double x = radius * Math.Sin(bearingAngle);
      double z = radius * Math.Cos(bearingAngle);
      double y = (b - 1) * layerHeight + (c - 1) *
         layerHeight;

      //T1.addMember(sNode, eNode, 2);

      //TowerNodes[b-1, a-1] = T1.addNode(x, y, z);
```

```
      TowerNodes[levelNum, a - 1] = T1.addNode(x, y, z)
          ;
    }

    levelNum++;
  }
}

for (int b = 0; b < layerCount * 2; b += 2)
{
  for (int a = 0; a < strutCount; a++)
  {
    int endStrutNum = a + strutCount - 2;
    endStrutNum = endStrutNum % strutCount;

    int endStrutNum2 = a + strutCount - 1;
    endStrutNum2 = endStrutNum2 % strutCount;

    /*
    if (endStrutNum >= strutCount)
    {
      endStrutNum -= strutCount;
    }
    else if (endStrutNum < 0)
    {
      endStrutNum += strutCount;
    }
    */

    //struts
    T1.addMember(TowerNodes[b, a], TowerNodes[b + 1,
        endStrutNum], 2);

    //vertical cables
    Member Mv = T1.addMember(TowerNodes[b, a],
        TowerNodes[b + 1, endStrutNum2], 1);
    Mv.memberLabel = "vertical";
```

```
//Mv.SetTension(500);
//Mv.CalcNodeDirections();
//Mv.CalcCurrentLength();
//Mv.customLength = Mv.CurrentLength.Magnitude() *
   0.98;
//Mv.SetLength(0.260);

//prestress cable
/*    Model   Factor
 *    4 Strut   0.91 / 0.87
 *
 *
 *
 */
//Mv.SetLengthFactor(0.90);

//double baseAndTopCableLength = 0.200;

//base
if (b == 0)
{
  //base cables
  Member Mb = T1.addMember(TowerNodes[b, a],
     TowerNodes[b, endStrutNum2], 1);
  Mb.memberLabel = "base";
  //Mb.SetLength(baseAndTopCableLength);
}

//top of tower
if (b == layerCount * 2 - 2)
{
  //top cables
  Member Mt = T1.addMember(TowerNodes[b + 1, a],
     TowerNodes[b + 1, endStrutNum2], 1);
  Mt.memberLabel = "top";
  //Mt.SetLength(baseAndTopCableLength);
```

```
      //NOTE: SET THIS IN BuildModel() FUNCTION
      //apply INITIAL displacement to top node
      //TowerNodes[b + 1, a].Displacement.Y = -0.001;
         //1mm DOWN
      //TowerNodes[b + 1, a].Displacement.Y = 0.001;
         //1mm UP
    }
  }
}

//cables joining modules (Neither top nor base of tower
   )
for (int b = 1; b < layerCount * 2 - 1; b += 2)
{
  for (int a = 0; a < strutCount; a++)
  {
    int endStrutNum = a + strutCount + 1;
    endStrutNum = endStrutNum % strutCount;

    int endStrutNum2 = a + strutCount;
    endStrutNum2 = endStrutNum2 % strutCount;

    T1.addMember(TowerNodes[b, a], TowerNodes[b + 1,
       endStrutNum], 1);
    T1.addMember(TowerNodes[b, a], TowerNodes[b + 1,
       endStrutNum2], 1);
  }
}

//support conditions at base
for (int a = 0; a < strutCount; a++)
{
  //fix all in y only
  TowerNodes[0, a].Fixity = new Vector(0, 1, 0);

  //fix all
  //TowerNodes[0, a].Fixity = new Vector(1, 1, 1);
```

```
        }

        //fix 1st node in all directions
        //TowerNodes[0, 0].Fixity = new Vector(1, 1, 1);

        //fix 2nd node in all directions
        //TowerNodes[0, 1].Fixity = new Vector(1, 1, 1);

        //top nodes
        for (int a = 0; a < strutCount; a++)
        {
          TowerNodes[layerCount * 2 - 2 + 1, a].nodeLabel = "
              top";
          //TowerNodes[layerCount * 2 - 2 + 1, a].ForceApplied.
              Y = -150;
        }
      }

    }
  }
```

A.7 User Interface Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using System.Windows.Media.Media3D;
//using System.Windows.Forms.DataVisualization.Charting;

namespace Tensegrity_01
{
  public partial class Form1 : Form
  {
    _3D_Drawing_Control.UserControl1 dwgControl = new
        _3D_Drawing_Control.UserControl1();

    Tensegrity T1 = new Tensegrity();

    public bool stopIterations = false;

    //file writer
    System.IO.StreamWriter objWriter;

    public Form1()
    {
      InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
      //load 3d drawing control and draw axes
```

```
elementHost1.Child = dwgControl;

//Set Chart Stuff
chtEnergy.Series[0].Name = "Energy - J";
chtEnergy.Series[0].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Line;
chtEnergy.Series.Add("Energy Peaks");
chtEnergy.Series[1].BorderColor = Color.Red;
chtEnergy.Series[1].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Point;
chtEnergy.Series[1].Points.Add(new System.Windows.Forms
   .DataVisualization.Charting.DataPoint(0,0));
chtEnergy.Series.Add("Average");
chtEnergy.Series[2].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Line;
chtEnergy.Series[2].Color = Color.DarkGreen;

//Residuals
chtResidual.Series[0].Name = "Residual - X";
chtResidual.Series[0].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Line;
chtResidual.Series.Add("Y");
chtResidual.Series[1].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Line;
chtResidual.Series[1].Color = Color.Red;
chtResidual.Series.Add("Z");
chtResidual.Series[2].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Line;
chtResidual.Series[2].Color = Color.DarkGreen;

chtDispl.Series[0].Name = "Displacement (AN - mm)";
chtDispl.Series[0].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Line;

chtResults.Series[0].Name = "Results (F vs D)";
chtResults.Series[0].ChartType = System.Windows.Forms.
   DataVisualization.Charting.SeriesChartType.Line;
```

```
  //create file for storing energies
  string fileName = "energies.csv";
  objWriter = new System.IO.StreamWriter(fileName);
  objWriter.WriteLine("Iteration, Energy, Lambda");

  //get inputs for this tensegrity model
  lstModels.SelectedIndex = 16;
  //BuildStrut2();
  //BuildPendulum();
  //BuildCableCantilever2();
  //BuildCableCantilever3();
  //BuildSnelsonX();
  //BuildPrism();
  //BuildBeam();

  //Modules
  //BuildSquarePrism(new Point3D());

  chkKineticDamping.Checked = T1.kineticDamping;
  chkTensegritySystem.Checked = T1.tensegritySystem;
  //updwnTimeIncrement.Value = T1.timeIncrement;
  //updwnLambda.Value = (int)T1.lambda;

  T1.resolveMembersToNodes();
  btnDisplay.PerformClick();

  RepositionFormElements();
}

private void btnGo_Click(object sender, EventArgs e)
{
  Iterate();
  lstEnergy.Items.Insert(0, T1.iterations + ". " + T1.
     kineticEnergy);
}
```

```
private void Iterate()
{
  btnRun.PerformClick();

  //if (chkUpdateDrawing.Checked)
  {
    btnDisplay.PerformClick();
  }
}

private void btnRun_Click(object sender, EventArgs e)
{
  RunIteration(true);
}

private void RunIteration(bool showProgress)
{
  T1.kineticDamping = chkKineticDamping.Checked;
  T1.tensegritySystem = chkTensegritySystem.Checked;
  T1.Run();

  if (showProgress)
  {
    T1.StoreKineticEnergyData(lstEnergy, objWriter);

    //add point to Energy Chart
    try
    {
      if (T1.kineticEnergy < double.MaxValue && T1.
          kineticEnergy > double.MinValue)
      {
        //Energy Value
        chtEnergy.Series[0].Points.Add(new System.Windows
            .Forms.DataVisualization.Charting.DataPoint(T1
            .iterations, T1.kineticEnergy));

        //Energy Peak
```

```
if (chtEnergy.Series[1].Points[chtEnergy.Series
    [1].Points.Count - 1].XValue != T1.
    lastDampedIterationNum)
{
  chtEnergy.Series[1].Points.Add(new System.
      Windows.Forms.DataVisualization.Charting.
      DataPoint(T1.lastDampedIterationNum, T1.
      kineticEnergy));
}

//Energy Average
chtEnergy.Series[2].Points.Add(new System.Windows
    .Forms.DataVisualization.Charting.DataPoint(T1
    .iterations, T1.kineticEnergyAvg));

//residuals
chtResidual.Series[0].Points.Add(new System.
    Windows.Forms.DataVisualization.Charting.
    DataPoint(T1.iterations, T1.nodes[T1.
    activeNode].ForceResidual.X));
chtResidual.Series[1].Points.Add(new System.
    Windows.Forms.DataVisualization.Charting.
    DataPoint(T1.iterations, T1.nodes[T1.
    activeNode].ForceResidual.Y));
chtResidual.Series[2].Points.Add(new System.
    Windows.Forms.DataVisualization.Charting.
    DataPoint(T1.iterations, T1.nodes[T1.
    activeNode].ForceResidual.Z));

//Displacement
double displ = T1.nodes[T1.activeNode].
    Displacement.Y * 1000;
chtDispl.Series[0].Points.Add(new System.Windows.
    Forms.DataVisualization.Charting.DataPoint(T1.
    iterations, displ));
lblDisplacement.Text = "Displacement: " + Math.
    Round(displ, 3) + " mm";
```

```
      }
      else
      {
        lstEnergy.Items.Insert(0, "Energy too high to
            insert point on chart");
      }
    }
    catch
    {
      lstEnergy.Items.Insert(0, "Error adding point to
          graph");
    }
  }
}

//shows node and member details in listview and draws
    system
private void btnDisplay_Click(object sender, EventArgs e)
{
  //T1.AddConfigToTreeview(tvwConfig);
  T1.addNodesToListview(lvwNodes);
  T1.addMembersToListview(lvwMembers);
  //this.Text = T1.lambda.ToString();
  //updwnLambda.Value = (int)T1.lambda;
  txtLambda.Text = T1.lambda.ToString();

  //draw
  dwgControl.ClearViewport();
  //dwgControl.DrawAxes();
  T1.SetDrawingControl(dwgControl);
  T1.Draw(true, chkDrawNodes.Checked, chkDrawForces.
      Checked);
}

private void btnSimulate_Click(object sender, EventArgs e
    )
{
```

```
  if (btnSimulate.Text == "Simulate")
  {
    //chtDispl.Series.Insert(0, new System.Windows.Forms.
       DataVisualization.Charting.Series());
    //chtDispl.Series[0].ChartType = System.Windows.Forms
       .DataVisualization.Charting.SeriesChartType.Line;

    tbMain.SelectedIndex = 4;
    stopIterations = false;
    btnSimulate.Text = "Stop";
    //btnSimulate.Enabled = false;
    pbIterations.Maximum = (int)updwnIterations.Value;

    SimulateIterations((int)updwnIterations.Value, true);

    btnSimulate.Text = "Simulate";
    btnSimulate.Enabled = true;
  }
  else
  {
    stopIterations = true;
  }

  btnDisplay.PerformClick();
  lstEnergy.Items.Insert(0, T1.iterations + ". " + T1.
     kineticEnergy);

  if (T1.nodes[T1.activeNode] != null)
  {
    lstEnergy.Items.Insert(0, "    Displ: " + Math.Round(
       T1.nodes[T1.activeNode].Displacement.Y * 1000, 3)
       + "mm");
  }
}

void SimulateIterations(int count, bool showProgress)
{
```

```
for (int i = 0; i < count; i++)
{
  RunIteration(showProgress);

  if (showProgress)
  {
    if (chkUpdateDrawing.Checked)
    {
      btnDisplay.PerformClick();
    }

    pbIterations.Value = i + 1;
    try
    {
      Application.DoEvents();
    }
    catch (Exception e)
    {
      lstEnergy.Items.Insert(0, e.ToString());
    }
  }

  //check kineticEnergy Limit (Ceiling)
  if (T1.kineticEnergy > T1.kineticEnergyLimit)
  {
    lstEnergy.Items.Insert(0, "Energy too high (imposed
        limit reached)");
    break;
  }

  if (stopIterations)
  {
    lstEnergy.Items.Insert(0, "User Stopped");
    break;
  }
}
```

```
  //MessageBox.Show("Done");
}

private void updwnIterations_ValueChanged(object sender,
    EventArgs e)
{
  //btnSimulate.Text = "Simulate " + updwnIterations.
      Value + " Iterations";
}

private void btnClear_Click(object sender, EventArgs e)
{
  lvwNodes.Items.Clear();
  lvwMembers.Items.Clear();
  //lstEnergy.Items.Clear();
  chtEnergy.Series[0].Points.Clear();

  //KE peaks graph
  chtEnergy.Series[1].Points.Clear();
  chtEnergy.Series[1].Points.Add(new System.Windows.Forms
      .DataVisualization.Charting.DataPoint(T1.iterations,
       0));
  T1.lastDampedIterationNum = T1.iterations;
  chtEnergy.Series[2].Points.Clear();

  chtResidual.Series[0].Points.Clear();
  chtResidual.Series[1].Points.Clear();
  chtResidual.Series[2].Points.Clear();

  chtDispl.Series[0].Points.Clear();
}

private void Form1_FormClosed(object sender,
    FormClosedEventArgs e)
{
  objWriter.Close();
}
```

```
private void chkKineticDamping_CheckedChanged(object
    sender, EventArgs e)
{
  T1.kineticDamping = chkKineticDamping.Checked;
}

private void chkTensegritySystem_CheckedChanged(object
    sender, EventArgs e)
{
  T1.tensegritySystem = chkTensegritySystem.Checked;
}

private void btnSimulate100_Click(object sender,
    EventArgs e)
{
  pbIterations.Maximum = 100;

  for (int i = 0; i < 100; i++)
  {
    Iterate();

    //if (i % 25 == 0)
    //{
      pbIterations.Value = i + 1;
      Application.DoEvents();
    //}
  }

  btnDisplay.PerformClick();
  MessageBox.Show("Done");
}

private void lstModels_SelectedIndexChanged(object sender
    , EventArgs e)
{
  T1 = new Tensegrity();
```

```
  btnClear.PerformClick();

  BuildModel(lstModels.SelectedIndex);

  btnInitialUpdate.PerformClick();
  btnDisplay.PerformClick();
}

private void btnRedraw_Click(object sender, EventArgs e)
{
  T1.resolveMembersToNodes();
  btnDisplay.PerformClick();
}

private void btnCurrentSave_Click(object sender,
   EventArgs e)
{
  saveFileDialog1.DefaultExt = "ten";
  saveFileDialog1.FileName = "model.ten";
  DialogResult Dr = saveFileDialog1.ShowDialog();

  if (Dr.ToString() == "OK")
  {
    string fileName = saveFileDialog1.FileName;
    T1.Write(fileName);
  }
}

private void btnCurrentLoad_Click(object sender,
   EventArgs e)
{
  DialogResult Dr = openFileDialog1.ShowDialog();

  if (Dr.ToString() == "OK")
  {
    string fileName = openFileDialog1.FileName;
    T1 = new Tensegrity();
```

```
    T1.Read(fileName);
    btnRedraw.PerformClick();
    btnDisplay.PerformClick();
    btnInitialUpdate.PerformClick();
  }
}

private void chkDrawNodes_CheckedChanged(object sender,
   EventArgs e)
{
  btnDisplay.PerformClick();
}

private void chkDrawForces_CheckedChanged(object sender,
   EventArgs e)
{
  btnDisplay.PerformClick();
}

private void btnSetAsInitial_Click(object sender,
   EventArgs e)
{
  T1.SetCurrentConfigAsInitialConfig();
  btnDisplay.PerformClick();
}

private void btnInitialUpdate_Click(object sender,
   EventArgs e)
{
  T1.AddConfigToTreeview(tvwConfig);
}

private void tvwConfig_AfterLabelEdit(object sender,
   NodeLabelEditEventArgs e)
{
  // TODO: ERROR HANDLING HERE
  //TreeNode selNode = tvwConfig.SelectedNode;
```

```
if (e.CancelEdit == false)
{
  if (e.Node.Level == 2) // Model Property, Node or
     Member
  {
    // model property
    if (e.Node.Parent.Parent.Text == "Model")
    {
      if (e.Node.Parent.Text == "Time Increment")
      {
        T1.timeIncrement = Convert.ToDouble(e.Label);
      }

      if (e.Node.Parent.Text == "Lambda")
      {
        T1.lambda = Convert.ToDouble(e.Label);
      }
    }
    else if (e.Node.Parent.Parent.Text == "Nodes") //
       node
    {
      int nodeNumber = (int)e.Node.Parent.Parent.Index
         + 1;

    }
  }
  else if (e.Node.Level == 3) // Node or Member
     Variable
  {
    // node
    if (e.Node.Parent.Parent.Parent.Text == "Nodes")
    {
      int nodeNumber = (int)e.Node.Parent.Parent.Index
         + 1;
      int varIndex = e.Node.Index; // X,Y or Z (0,1,2)
```

```
if (e.Node.Parent.Name == "Coordinates")
{
  if (varIndex == 0) //X
  {
    T1.nodes[nodeNumber].X = Convert.ToDouble(e.
        Label);
  }
  else if (varIndex == 1) //Y
  {
    T1.nodes[nodeNumber].Y = Convert.ToDouble(e.
        Label);
  }
  else if (varIndex == 2) //Z
  {
    T1.nodes[nodeNumber].Z = Convert.ToDouble(e.
        Label);
  }
}
else if (e.Node.Parent.Name == "Fixity")
{
  if (varIndex == 0) //X
  {
    T1.nodes[nodeNumber].Fixity.X = Convert.
        ToDouble(e.Label);
  }
  else if (varIndex == 1) //Y
  {
    T1.nodes[nodeNumber].Fixity.Y = Convert.
        ToDouble(e.Label);
  }
  else if (varIndex == 2) //Z
  {
    T1.nodes[nodeNumber].Fixity.Z = Convert.
        ToDouble(e.Label);
  }
}
else if (e.Node.Parent.Name == "Force Applied")
```

```
        {
          if (varIndex == 0) //X
          {
            T1.nodes[nodeNumber].ForceApplied.X = Convert
                .ToDouble(e.Label);
          }
          else if (varIndex == 1) //Y
          {
            T1.nodes[nodeNumber].ForceApplied.Y = Convert
                .ToDouble(e.Label);
          }
          else if (varIndex == 2) //Z
          {
            T1.nodes[nodeNumber].ForceApplied.Z = Convert
                .ToDouble(e.Label);
          }
        }
      }
      else if (e.Node.Parent.Parent.Parent.Text == "
          Members")
      {

      }
    }
  }

  btnDisplay.PerformClick();
}

private void tvwConfig_Click(object sender, EventArgs e)
{

}

private void tvwConfig_NodeMouseClick(object sender,
    TreeNodeMouseClickEventArgs e)
{
```

```
  TreeNode selNode = e.Node; //tvwConfig.SelectedNode;

  if (selNode != null)
  {
    if (selNode.Level == 1) // Model Properties, Node or
        Member
    {
      //node
      if (selNode.Parent.Text == "Nodes")
      {
        int nodeNumber = (int)selNode.Index + 1;
        T1.selectedNodeNum = nodeNumber;
      }
      //member
      else if (selNode.Parent.Text == "Members")
      {
        int memNumber = (int)selNode.Index + 1;
        T1.selectedMemberNum = memNumber;
      }
    }
  }

  btnDisplay.PerformClick();
}

private void Form1_Resize(object sender, EventArgs e)
{
  RepositionFormElements();
}

private void RepositionFormElements()
{
  tbMain.Top = this.Height - tbMain.Height - 40;
  tbControls.Top = tbMain.Top;
  elementHost1.Width = this.Width - 30;
  elementHost1.Height = this.Height - tbMain.Height - 60;
```

```
  btnViewportSize.Top = elementHost1.Top + elementHost1.
     Height - btnViewportSize.Height;
  btnViewportSize.Left = elementHost1.Left + elementHost1
     .Width - btnViewportSize.Width;
  btnSaveBitmap.Top = btnViewportSize.Top;
  btnSaveBitmap.Left = btnViewportSize.Left -
     btnSaveBitmap.Width - 8;
}

private void btnSaveBitmap_Click(object sender, EventArgs
    e)
{
  //string fileName = "bitmaps/model " + T1.iterations +
     ".bmp";

  saveFileDialog1.FileName = "model.bmp";
  saveFileDialog1.DefaultExt = "bmp";
  DialogResult Dr = saveFileDialog1.ShowDialog();

  if (Dr.ToString() == "OK")
  {
    Bitmap bm = new Bitmap(elementHost1.Width,
       elementHost1.Height);
    Rectangle r = new Rectangle(0, 0, elementHost1.Width,
        elementHost1.Height);
    elementHost1.DrawToBitmap(bm, r);

    //System.Windows.Forms.Integration.ElementHost eh =
       new System.Windows.Forms.Integration.ElementHost()
       ;
    //dwgControl.dra

    string fileName = saveFileDialog1.FileName;
    bm.Save(fileName);
  }
}
```

```
private void btnViewportSize_Click(object sender,
    EventArgs e)
{
  elementHost1.Height = Form1.ActiveForm.Height - 60;
  btnViewportSize.Top = elementHost1.Top + elementHost1.
      Height - btnViewportSize.Height;
  btnViewportSize.Left = elementHost1.Left + elementHost1
      .Width - btnViewportSize.Width;
  btnSaveBitmap.Top = btnViewportSize.Top;
  btnSaveBitmap.Left = btnViewportSize.Left -
      btnSaveBitmap.Width - 8;
}

private void btnViewPersp_Click(object sender, EventArgs
    e)
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(10, 10, 10));
  dwgControl.SetCameraLookDirection(new Vector3D(-1, -1,
      -1));
}

private void btnViewTop_Click(object sender, EventArgs e)
{
  //set camera
  dwgControl.SetCameraPosition(new Point3D(10, 10, 10));
  dwgControl.SetCameraLookDirection(new Vector3D(0, -1,
      0));
}

private void updwnTimeIncrement_ValueChanged(object
    sender, EventArgs e)
{
  T1.timeIncrement = Math.Pow(10, -(int)
      updwnTimeIncrement.Value);
}
```

```
private void btnIncrementalSolve_Click(object sender,
    EventArgs e)
{
  updwnIterations.Value = 20000;

  int lowestExponent = 2; // i.e. timeIncrement = 1E-5
  int highestExponent = 2; //16;

  for (int a = lowestExponent; a <= highestExponent; a++)
  {
    updwnIterations.Value /= 2;

    updwnTimeIncrement.Value = a;
    btnSimulate.PerformClick();

    if (stopIterations)
    {
      break;
    }
  }
}

private void btnSolveAndOutput_Click(object sender,
    EventArgs e)
{
  double initialDispl = 0;
  btnSimulate.Text = "Stop";
  stopIterations = false;

  int nodeNumber = T1.activeNode; //2; //16 // the node
     we are working with (i.e. applying the force to)
  //nodeNumber = 8; //nodenumber not saved!!

  //apply force in increments
  Vector ForceTotal = T1.nodes[nodeNumber].ForceApplied;
  //int incrementCount = Math.Abs((int)(forceTotal / 15))
     ; //N per increment
```

```
int incrementCount = 10;
int iterationCount = 10000;
double lambdaStart = 500;
double lambdaIncreaseFactor = 10; // 1 = no increase
double lambdaMax = 50000;
double allowableDisplError = 1e-5; //(metres) allowable
    error in displacment (see below)
double allowableEnergyError = 1E-14;
int continueNum = 0;
int continueMax = 10; //maximum number of times to
   continue iterations when error requirements not met

//T1.lambdaInitial = -forceTotal / incrementCount;

string fileName;
fileName = "C:/Muz/UCT2010/Thesis/Tensegrity Designer
   Software/new results/results - " + T1.name + ".csv";

System.IO.StreamWriter objWriter3 = new System.IO.
   StreamWriter(fileName, false);
objWriter3.WriteLine("Initial Displacement,," + T1.
   nodes[nodeNumber].Displacement.Y);
objWriter3.WriteLine("No., Force, Disp X, -Y, Z");

//PB for increments
pbSolve.Maximum = incrementCount;

//PB for iterations
pbIterations.Maximum = iterationCount;

//Results Graph
chtResults.Series[0].Points.Clear();

for (int a = 0; a <= incrementCount; a++)
{
  //reset to initial config
  //T1.ResetConfig();
```

```
T1.lambda = lambdaStart;
//T1.peakCount = 0;

//set force
double forceFraction = (double)a / (double)
    incrementCount;
Vector CurrentForce = ForceTotal.multiplyByValue(
    forceFraction);

//if (a > incrementCount / 2)
//{
//  CurrentForce = CurrentForce.multiplyByValue(2);
//}

//Apply force factor (fraction)
T1.SetNodesAppliedForceFactor(forceFraction);

//T1.setNodeAppliedForce("top", CurrentForce);

//T1.nodes[nodeNumber].ForceApplied.Y = currentForce;

//T1.setNodeAppliedVerticalForce("top", currentForce)
    ;

//graph
btnClear.PerformClick();
//chtDispl.Series[0].Points.Clear();
//chtDispl.Series.Insert(0, new System.Windows.Forms.
    DataVisualization.Charting.Series());
//T1.iterations = 0;

//info
lstEnergy.Items.Insert(0, "====Solving Config. for "
    + CurrentForce.Magnitude() + "N. Pass " + a + " of
     " + incrementCount);
```

```
//perform calcs (solve)
//btnIncrementalSolve.PerformClick();
double prevDispl = T1.nodes[nodeNumber].Displacement.
    Y;
double deltaDispl;

bool continueSolving = true;

continueNum = 0;

do
{
  prevDispl = T1.nodes[nodeNumber].Displacement.Y;

  updwnIterations.Value = iterationCount;
  SimulateIterations(iterationCount, false);

  //refresh
  Application.DoEvents();

  if (stopIterations)
  {
    goto incrementEnd;
  }

  deltaDispl = Math.Abs(prevDispl - T1.nodes[
      nodeNumber].Displacement.Y);
  lstEnergy.Items.Insert(0, "Displ: " + Math.Round(T1
      .nodes[nodeNumber].Displacement.Y * 1000,3) + "
      mm Change: " + Math.Round(deltaDispl*1000,3) + "
      mm");
  //lstEnergy.Items.Insert(0, "  KE: " + T1.
      kineticEnergy);
  //lstEnergy.Items.Insert(0, "  Lambda: " + T1.
      lambda);

  double energyDiff = T1.kineticEnergy -
```

```
    allowableEnergyError;
  //lstEnergy.Items.Insert(0, "KE Diff. from alwbl: "
      + energyDiff);

  if (T1.kineticEnergy > allowableEnergyError)
  {
    T1.lambda *= lambdaIncreaseFactor;

    if (T1.lambda > lambdaMax)
    {
      T1.lambda = lambdaMax;
    }

    lstEnergy.Items.Insert(0, "Target KE not met,
       increased lambda: " + T1.lambda);
  }
  else
  {
    continueSolving = false;
  }

  continueNum++;

  if (continueNum >= continueMax)
  {
    lstEnergy.Items.Insert(0, "Maximum No. of
     continues reached (" + continueMax + "), going
       to next iteration");
    break;
  }

//} while (continueSolving || deltaDispl >
   allowableDisplError);

} while (continueSolving);
//} while (deltaDispl > allowableDisplError);
//} while (T1.kineticEnergy > allowableEnergyError);
```

```
    //} while (T1.kineticEnergy > allowableEnergyError ||
        deltaDispl > allowableDisplError);

    //show displacement
    if (a == 0)
    {
      //initial equilibrium displacement
      initialDispl = T1.nodes[nodeNumber].Displacement.Y;
    }
    double curDispl = Math.Round(T1.nodes[nodeNumber].
       Displacement.Y * 1000, 3);
    double actDispl = Math.Round((T1.nodes[nodeNumber].
       Displacement.Y - initialDispl) * 1000, 3);
    lstEnergy.Items.Insert(0, "Displ. at node " +
       nodeNumber + " = " + curDispl + "mm  [" + actDispl
        + "mm]");

    //show results on graph
    chtResults.Series[0].Points.Add(new System.Windows.
       Forms.DataVisualization.Charting.DataPoint(-
       curDispl, CurrentForce.Magnitude()));

    //store results
    objWriter3.WriteLine(a + ", " + CurrentForce.
       Magnitude() + ", "
      + T1.nodes[nodeNumber].Displacement.X + ", "
      + -T1.nodes[nodeNumber].Displacement.Y + ", "
      + T1.nodes[nodeNumber].Displacement.Z);

    pbSolve.Value = a;
  }

incrementEnd: ;

  lstEnergy.Items.Insert(0, "    Final Displ: " + Math.
     Round(T1.nodes[nodeNumber].Displacement.Y * 1000, 3)
     );
```

```
  lstEnergy.Items.Insert(0, "Done Solving");
  btnDisplay.PerformClick();

  //reset force
  //T1.nodes[nodeNumber].ForceApplied.Y = forceTotal;
  T1.setNodeAppliedForce("top", ForceTotal);

  objWriter3.Close();

  btnSimulate.Text = "Simulate";
}

private void updwnLambda_ValueChanged(object sender,
    EventArgs e)
{
  T1.lambda = (int)updwnLambda.Value;
}

private void btnApplyLoad_Click(object sender, EventArgs
    e)
{
  // 4 Strut:
  T1.setNodeAppliedVerticalForce("top", -150);

  btnDisplay.PerformClick();
}

private void btnSaveCoords_Click(object sender, EventArgs
    e)
{
  saveFileDialog1.DefaultExt = "txt";
  saveFileDialog1.FileName = "nodes and members";
  DialogResult Dr = saveFileDialog1.ShowDialog();

  if (Dr.ToString() == "OK")
  {
    string fileName = saveFileDialog1.FileName;
```

```
    T1.SaveForAbaqus(fileName);
  }
}

private void txtLambda_TextChanged(object sender,
   EventArgs e)
{
  try
  {
    T1.lambda = Convert.ToDouble(txtLambda.Text);
  }
  catch
  {

  }
}

private void btnTesting_Click(object sender, EventArgs e)
{
  //testing strut count vs stiffness
  stopIterations = false;

  // strut vs stiffness
  /*
  for (int a = 3; a<=7; a++)
  {
    if (stopIterations)
    {
      break;
    }

    T1 = new Tensegrity();
    T1.name = a + " strut module";
    BuildTower(a, 1, 0.260, 0.15);
    T1.activeNode = a*2;
    T1.setTotalAppliedVerticalForce(-500);
    btnSolveAndOutput.PerformClick();
```

```
}
*/

/*
//height vs layers
double totalHeight = 1.5;
int strutCount = 4;


for (int a = 1; a <= 5; a++)
{
  if (stopIterations)
  {
    break;
  }

  T1 = new Tensegrity();
  T1.name = a + " layer tower";
  double layerHeight = totalHeight / a;
  BuildTower(strutCount, a, layerHeight, 0.15);
  T1.activeNode = T1.nodes.Length - 1;
  T1.setTotalAppliedVerticalForce(-500);
  btnSolveAndOutput.PerformClick();
}
*/

//tower pre-stress variation
double maxLengthFactor = 1;
double minLengthFactor = 0.99;
int total = 10;

for (int a = 1; a <= total; a++)
{
  this.Text = "Run " + a + " of " + total;
  if (stopIterations)
  {
    break;
```

```
            }

            T1 = new Tensegrity();
            T1.name = "tower prestress " + a;

            BuildTower(6, 3, 0.340, 0.15);
            T1.activeNode = 31;
            T1.setNodeAppliedVerticalForce("top", -100);

            double lengthFactor = minLengthFactor + (
                maxLengthFactor - minLengthFactor) * ((double)a /
                (double)total);
            T1.setMemberLength("vertical", 0.349 * lengthFactor);

            //btnSolveAndOutput.PerformClick();
          }
        }

        private void btnClearList_Click(object sender, EventArgs
            e)
        {
          lstEnergy.Items.Clear();
        }
      }

}
```

A.8 Program Initialisation Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace Tensegrity_01
{
  static class Program
  {
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new Form1());
    }
  }
}
```

Appendix B

ABAQUS Input File for Four Strut Module


```
*Heading
** Job name: Job-1 Model name: Model-1
** Generated by: Abaqus/CAE Version 6.8-1
*Preprint, echo=NO, model=NO, history=NO, contact=NO
**
** PARTS
**
*Part, name=Full
*Node
      1, -0.150000006,           0.,           0.
      2,  0.105999999,   0.25999999, -0.105999999
      3,  0.105999999,   0.25999999,  0.105999999
      4, -0.105999999,   0.25999999, -0.105999999
      5,           0.,           0., -0.150000006
      6, -0.105999999,   0.25999999,  0.105999999
      7,           0.,           0.,  0.150000006
      8,  0.150000006,           0.,           0.
*Element, type=T3D2
1, 1, 2
2, 3, 2
```

```
3, 1, 4
4, 5, 2
5, 6, 3
6, 4, 6
7, 1, 7
8, 2, 4
 9, 5, 1
10, 5, 3
11, 8, 6
12, 7, 4
13, 8, 5
14, 8, 3
15, 7, 6
16, 7, 8
*Node
      9,           0.,           0.,           0.
*Nset, nset=Full-RefPt_, internal
9,
*Nset, nset=Wire-1-Set-1
 1, 5, 7, 8
*Elset, elset=Wire-1-Set-1
  7,  9, 13, 16
*Nset, nset=TopSquare
 2, 3, 4, 6
*Elset, elset=TopSquare
 2, 5, 6, 8
*Nset, nset=Wire-3-Set-1, generate
 1,  8,  1
*Elset, elset=Wire-3-Set-1
  3,  4, 14, 15
*Nset, nset=_PickedSet8, internal, generate
 1,  8,  1
*Elset, elset=_PickedSet8, internal
  2,  3,  4,  5,  6,  7,  8,  9, 13, 14, 15, 16
*Nset, nset=Wire-4-Set-1, generate
 1,  8,  1
*Elset, elset=Wire-4-Set-1
```

```
  1, 10, 11, 12
*Nset, nset=_PickedSet11, internal, generate
 1,  8,  1
*Elset, elset=_PickedSet11, internal
  1, 10, 11, 12
*Nset, nset=TopNode
 3,
*Nset, nset=VertCable
 3, 8
*Elset, elset=VertCable
 14,
** Section: StrutSection
*Solid Section, elset=_PickedSet11, material=Aluminium
3.55314e-05,
** Section: CableSection
*Solid Section, elset=_PickedSet8, material="Stainless Steel"
7.85398e-07,
*End Part
**
**
** ASSEMBLY
**
*Assembly, name=Assembly
**
*Instance, name=Full-1, part=Full
*End Instance
**
*Nset, nset=_PickedSet19, internal, instance=Full-1
 7,
*Nset, nset=_PickedSet21, internal, instance=Full-1
 5, 8
*Nset, nset=_PickedSet23, internal, instance=Full-1
 2, 3, 4, 6
*Nset, nset=_PickedSet24, internal, instance=Full-1
 1,
*Nset, nset=_PickedSet25, internal, instance=Full-1
 2, 3, 4, 6
```

```
*End Assembly
**
** MATERIALS
**
*Material, name=Aluminium
*Elastic
 6.9e+10, 0.3
*Material, name="Stainless Steel"
*Elastic
 1.93e+11, 0.3
**
** BOUNDARY CONDITIONS
**
** Name: BC-1 Type: Displacement/Rotation
*Boundary
_PickedSet19, 1, 1
_PickedSet19, 2, 2
_PickedSet19, 3, 3
** Name: BC-3 Type: Displacement/Rotation
*Boundary
_PickedSet21, 2, 2
** Name: BC-4 Type: Displacement/Rotation
*Boundary
_PickedSet24, 1, 1
_PickedSet24, 2, 2
**
   ----------------------------------------------------------------


**
** STEP: Step-1
**
*Step, name=Step-1, nlgeom=YES
*Static
0.01, 1., 1e-05, 1.
**
** BOUNDARY CONDITIONS
**
```

```
** Name: Displ Type: Displacement/Rotation
*Boundary
_PickedSet23, 2, 2, -0.001
**
** OUTPUT REQUESTS
**
*Restart, write, frequency=0
**
** FIELD OUTPUT: F-Output-1
**
*Output, field, variable=PRESELECT
**
** HISTORY OUTPUT: Displacements
**
*Output, history
*Node Output, nset=Full-1.TopNode
U2,
**
** HISTORY OUTPUT: Stress
**
*Element Output, elset=Full-1.VertCable
MISES, S11, S22, S33
*End Step
**
   ----------------------------------------------------------------


**
** STEP: Loading
**
*Step, name=Loading, nlgeom=YES
*Static
0.005, 1., 1e-05, 1.
**
** BOUNDARY CONDITIONS
**
** Name: BC-1 Type: Displacement/Rotation
*Boundary, op=NEW
```

```
_PickedSet19, 1, 1
_PickedSet19, 2, 2
_PickedSet19, 3, 3
** Name: BC-3 Type: Displacement/Rotation
*Boundary, op=NEW
_PickedSet21, 2, 2
** Name: BC-4 Type: Displacement/Rotation
*Boundary, op=NEW
_PickedSet24, 1, 1
_PickedSet24, 2, 2
** Name: Displ Type: Displacement/Rotation
*Boundary, op=NEW
**
** LOADS
**
** Name: Load-1   Type: Concentrated force
*Cload
_PickedSet25, 2, -200.
**
** OUTPUT REQUESTS
**
*Restart, write, frequency=0
**
** FIELD OUTPUT: F-Output-1
**
*Output, field, variable=PRESELECT
**
** HISTORY OUTPUT: Displacements
**
*Output, history
*Node Output, nset=Full-1.TopNode
CF2, U2
**
** HISTORY OUTPUT: Stress
**
*Element Output, elset=Full-1.VertCable
MISES, S11, S22, S33
```

```
*End Step
```