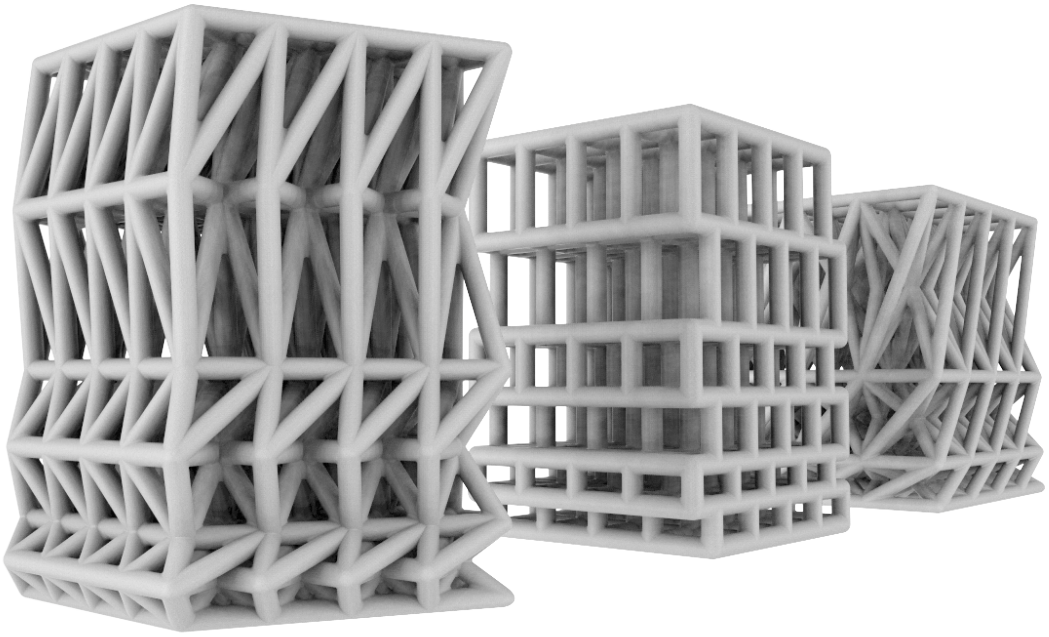
CHALMERS
UNIVERSITY OF TECHNOLOGY



Design Optimization for 3D Printed Energy Absorbing Structures Inspired by Nature

A theoretical geometry evaluation for maximizing specific energy absorption

Master's thesis in Material and Computational Mechanics

Alexander Olsson & Mattias Naarttijärvi

Department of Applied Mechanics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis 2017:34

Design Optimization for 3D Printed Energy Absorbing Structures Inspired by Nature

A theoretical geometry evaluation for maximizing specific energy absorption

Alexander Olsson
Mattias Naarttijärvi

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Applied Mechanics
Division of Material and Computational Mechanics
Chalmers University of Technology
Gothenburg, Sweden 2017

Design Optimization for 3D Printed Energy Absorbing Structures Inspired by Nature
A theoretical geometry evaluation for maximizing specific energy absorption
Alexander Olsson
Mattias Naarttijärvi

Supervisor: Spyros Tsampas, Swerea Sicomp
Examiner: Leif Asp, Department of Applied Mechanics, Chalmers University of Technology

Master's Thesis 2017:34
Department of Applied Mechanics
Division of Material and Computational Mechanics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Final result represented as four by four repeated unit columns.

Typeset in LaTeX
Printed by Department of Applied Mechanics
Gothenburg, Sweden 2017

Design Optimization for 3D Printed Energy Absorbing Structures Inspired by Nature
A theoretical geometry evaluation for maximizing specific energy absorption
Alexander Olsson
Mattias Naarttijärvi
Department of Applied Mechanics
Chalmers University of Technology

Abstract

Transportation is a major part of people's every day life in today's society allowing them to get to their jobs, commute, trade, travel etc. Motorcyclists and cyclists are among the most vulnerable road users and in case of an accident, they are highly dependent on bearing a helmet to protect against severe damage. Studies have shown that bearing a modern helmet provides 63% to 88% reduction of sustained head and severe brain injury in case of an accident for bicyclist. A route to further improve helmets, besides developing stiffer and tougher materials, is to develop a helmet that also relies on the material structure, i.e. its inner geometry and architecture, for energy absorption. Additive manufacturing or 3D printing allows three-dimensional objects or components to be manufactured with a complexity which would be difficult or near impossible to realize with today's conventional manufacturing techniques used for helmets. With the possibilities and precision 3D printing enables in mind, structures based on geometries found in nature is investigated and theoretically optimized to absorb as much energy as possible on impact meanwhile keeping the mass low.

Three main structures made up of beam elements were chosen and further investigated in a script. The script is designed to generate and optimize the structure by positioning its elements and varying their position, width, height and radius and evaluate it regarding specific energy absorption by doing a FEM analysis and a buckling analysis. Ultimately the script work as intended by successfully generate the sought structures and autonomously update the structures variables and return an optimized combination of the variables which maximized the structures ability to absorb energy on impact.

Keywords: Beam, helmet, energy absorption, 3D printing, optimization

Designoptimering för 3D-printade energiabsorberande strukturer inspirerade av naturen
En teoretisk utvärdering av geometri med avseende att maximera specifik energiabsorption
Alexander Olsson
Mattias Naarttijärvi
Avdelning för Tillämpad Mekanik
Chalmers Tekniska Högskola

Sammanfattning

Transport är en stor del av människors vardag i dagens samhälle vare sig det är för att kunna komma till sitt jobb, pendla, handla eller resa. Motorcyklister och cyklister är bland de mest utsatta trafikanterna och i händelse av en olycka är de mycket beroende av att ha hjälm på sig för att skydda mot allvarliga skador. Studier har visat att ha på sig en hjälm minskar chansen att få en allvarlig hjärnskada med 63% till 88% vid eventuell olycka för en cyklist. Ett sätt att förbättra hjälmarna förutom att ta fram bättre och starkare material är att utveckla en hjälm som är beroende av materialstrukturen i hjälmen, dvs dess inre geometri för att öka dess energiabsorption. Additiv tillverkning eller 3D-printning gör att även mycket komplexa komponenter och strukturer kan tillverkas som annars med dagens tillverkningstekniker för hjälmar skulle vara svåra eller näst intill omöjliga att realisera. Med de möjligheter och precision som 3D-printning tillför i åtanke undersöks geometrier och strukturer som förekommer naturligt i naturen med avseende att finna strukturer som teoretiskt optimeras för att absorbera så mycket energi som möjligt samtidigt som dess vikt hålls låg.

Tre huvudsakliga strukturer uppbyggda utav balkelement valdes och undersöktes vidare i ett datorskript. Skriptet är konstruerat för att generera och optimera strukturerna genom att skapa och placera dess element samt variera elementens position, bredd, höjd och radie samtidigt som strukturen utvärderas gällande dess förmåga att ta upp energi genom att göra en FEM-analys och en bucklingsanalys. Skriptet fungerade som förväntat genom att framgångsrikt generera de eftersökta strukturerna och autonomt uppdatera dess variabler samt returnera en optimerad kombination av variablerna vilka maximerar strukturens förmåga att absorbera specifik energi vid kollision.

Nyckelord: balk, hjälm, energiabsorption, 3D printing, optimering

Acknowledgements

This report is submitted to fulfill the requirement to the Master's degree at Chalmers University of Technology, Gothenburg and has been carried out in collaboration with SWERA SICOMP in Mölndal, Sweden.

The work was conducted during the spring semester of 2017.

It is with gratitude we acknowledge the help and support from our supervisor Leif Asp from Chalmers University of Technology for his genuine interest in the project and his resourceful feedback and help. We would also like to acknowledge and express our appreciation to our supervisors at SWEREA SICOMP, Spyros Tsampas and Vasan Churchill Srinivasan Chandrasekaran for their helpful feedback and giving us vast freedom and trust throughout the project. Lastly we would like thank Erik Svensson of the Material and Manufacturing Technology of Chalmers for his input and discussions regarding material characteristics.

Gothenburg, June 2017 Alexander Olsson & Mattias Naarttijärvi

Contents

List of Figures

List of Tables

1

Introduction and background

Helmets have been widely used by humans for many centuries in order to protect the brain from heavy impacts. Historically, soldiers have been using helmets in battles for a long time and as technology and weapons have developed the helmets composition and material have also changed. The evolution has gone from early usage when the helmets were made of leather and cloths to today where a new generation of ultra-high-molecular-weight polyethylene fibers (UHMWPE) are used in combat helmets to protect against ballistic impacts [1]. In civilian life, the advantages of helmets was recognized and used much later. Today helmets are used in a wide range, stretching from recreational activities and sports, dangerous work activities like mining and construction and also transportation such as bicycle and motorcycles. Motorcyclists and cyclists are among the most vulnerable road users. In Sweden around 2000 people get seriously injured every year in bicycles accidents, around 20-30 of these accidents are fatal [2].
Studies have shown that bearing helmets provide a 63% to 88% reduction of sustained head and severe brain injury for all ages of bicyclists. The core function of a helmet is that it absorbs mechanical energy from the impact, relieving the head from as much energy as possible, hence reducing the risk of brain and head injuries [3]. A route to further improve helmets, besides developing better and tougher materials, is to develop a helmet that is relying on the material structure, i.e. its inner geometry, for energy absorption. How the material in the helmet is structured is limited by available manufacturing techniques and materials. However, the manufacturing techniques are developing rapidly and with 3D printing it is now possible to manufacture complex geometries with very high precision. In nature there are many complex mineral-based and protein-based bio-composites designed to absorb and resist impact and crushing. With additive manufacturing (AM) or 3D printing now available it is motivated to look at the structures and organisms in nature, shaped by the millions of years of evolution to inspire new ways to structure the materials in helmets in order to make them safer. This further allows for individual customization through tailor made helmets specifically designed for the user and its use case.

1.1 Objectives

This thesis aims to generate bio-inspired material architectures for improved energy absorption with potential to improve helmets. New material architectures are identified via adjustable models to evaluate different layer thicknesses and densi-

ties. Sponge-like structures are common in nature and such geometry will be sought determining parameters such as wall thickness and node density. With a flexible model, multiple analyses can be made to evaluate patterns that make a structure energy absorbent.
The goal of this thesis is to provide direction and guidance for future research in order to realize micro-structural designs for improved energy absorption in helmets to make them safer. The work is expected to result in identification of energy absorbent material architectures and required material properties to construct better liners for helmets. Furthermore, the proposed material concepts are to be assessed for their processability with current and future 3D printing capabilities.

1.2 Challenges and limitations

The optimization problem is limited to the liner in a helmet, i.e. not the hard shell surrounding the outside. Each geometry case will be represented by unit cells stacked to form unit columns. These unit columns are repeatable and will generate the entire helmet, however not in this thesis. The unit columns are only to be investigated with symmetry boundary conditions on the sides, simply supported at the bottom face and an evenly distributed force applied on the top face. These symmetries represents a flat structure, and not the curvature seen in helmets. The force is calculated to represent the force of impact that a helmet is required to withstand, according to European standards. The load is vertical. Any shear forces that may occur in an actual helmet crash test are not taken into consideration. The material selection is limited to polymers that are currently 3D printable. The scale of each geometry and the beams constructing it is limited to the accuracy of the 3D printer in order to evaluate against physical tests for Swerea. The simulations are performed with Matlab constructed by an elastic finite element analysis and buckling models. Plastic analysis will not be evaluated due to its complexity and the time consuming computational procedure. Regarding nature's influence of the thesis, it is limited to a conceptually inspiring level of the geometry due to material differences and the lack of documentation for mechanical properties of bio-composite micro-structures. Lastly, the thesis is 30 credits and limited time wise between January and June of 2017.

2

Literature study and relevant research in the field

The literature review aims to investigate promising structures and geometries found in nature as well as structures found in helmets today. It is a limited study in which few organisms and materials with well known mechanical properties is considered and further investigated.

2.1 Personal protection - Helmets

Most of today's helmets are of similar design. They have a hard outer shell which is attached to an inner layer consisting of some sort of foam. The core function of the outer shell is to protect the head from sharp objects and to distribute the impact load over a lager area. The outer shell dissipates a significant amount of the the mechanical energy (34%). The inner foam, absorbs the mechanical energy from the impact and distributes it over a large area, reliving the head from as much load as possible [3]. The foam can be of many types, but expanded polystyrene (EPS) is a common choice in bicycle helmets [4]. A cross section of a typical bicycle is depicted in Figure 2.1.

Figure 2.1: Cross section of a Kali Avita Carbon XC helmet [5].

2.2 Foams

For expanded polystyrene (EPS) foams the properties of cellular solids depend on two separate sets of properties, the geometry (cell size, shape, density, material

distribution between cell edges and faces) and material properties. A typical compressive stress-strain curve of elastomeric foams is shown in Figure 2.2. The curve can be divided into three regions: I, shows linear elasticity at low stresses. II, is a wide collapse plateau and III is the densification where the stresses rise steeply [6].

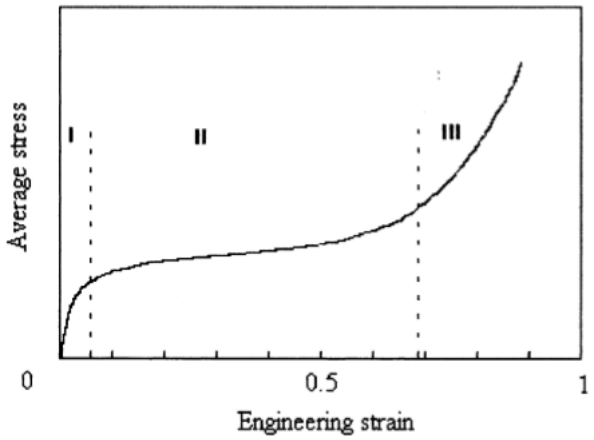


Figure 2.2: Compression stress-strain curve of a rigid foam [6].

The three stages has the following characteristics.

- I, The linear elasticity holds for small strains (3-5%) and consists of three types of strain: stretching of cell walls, bending of cell edges and compression of gas trapped into the cells. For the case of compressive load the plateau is associated with cell collapse due to the onset of plastic hinges. Opposing walls come into contact once the cells have almost completely collapsed, further compressive stresses arise leading to the final region of bottoming-out.
- II, Foams with a plastic yield point displays a ductile failure as well if loaded beyond their linear-elastic region. The plastic collapse results in a wide horizontal plateau in the stress-strain curve where the strains are no longer recoverable. This plastic deformation is exploited in energy-absorbing systems. The plastic collapse depends on three mechanisms: When the bending moment acting at cell walls exceed the allowable moment of the edges, there is an onset of permanent hinges, cell wall plastic stretching occurs and pressure of fluid contained into the cell increases.
- III, When cells are completely collapsed, at large compressive strains, the opposing walls are crushed together and the constituent material is compressed as well. As a consequence, the stress-strain curve rises steeply.

In Table 2.1 different experimental data on specific energy absorption for expanded polystyrene is shown. The test resulting in these data was a static compression test on different EPS densities and were performed according to free- and confined volume methods. With a confined volume method means that the foam is prevented to expand in a certain direction. In this experiment the foam was fitted in a cylindrical steel frame, preventing any radial expansion during the compression [6].

Table 2.1: Results of static compression tests on EPS [6].

	Specific Energy $[kJ/kg]$	
Nominal Density $[kg/m^3]$	Confined	Free
28	4.29	3.93
40	4.50	4.25
55	5.09	4.55
70	5.57	5.43

2.3 Energy absorbing structures in nature

In this chapter we present an overview of energy absorbing materiel structures in nature. The overview is not an extensive review of such materials but rather limited to some materials with anticipated high energy absorption.

2.3.1 Bones

In the human body, there are only two types of bone tissue: cortical and cancellous bone. Cortical bone is very dense and strong which makes it more difficult to fracture. Its primal function is to provide structural support for the body and its organs and tissues [7]. Cortical bone is structured by many microscopic cylinders called osteons. These cells produces bone matrices known as a lamellae [8]. A cross section of a human bone is depicted in Figure 2.3.

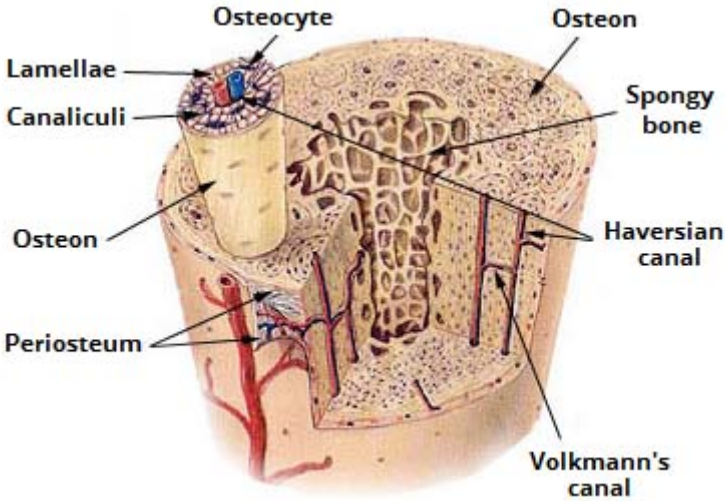


Figure 2.3: An illustrative cross section of a human bone [9].

Together the osteons forms a system of concentric circles which builds up the compact bone. When bundled, they form a strong support with high structural strength and rigidity [7]. The other type of bone, the cancellous bone, is located at the ends of the long bones, i.e. the cortical bones. In the typical adult human body, the cancellous bones make up about 20% of the skeleton. Although cancellous bone is

strong, it is more porous compared to cortical bones and thus more easily fractured. Cancellous bone is also known as spongy bone because of its similarity to a sponge or honeycomb. It has many open hollow spaces connected by flat planes of bone known as trabeculae [10]. Each time we move or do any physical activity, our hips, spine, and pelvis, is subjected to mechanical stresses. The strength that keeps the bones from breaking is provided by the trabeculae, see Figure 2.4.

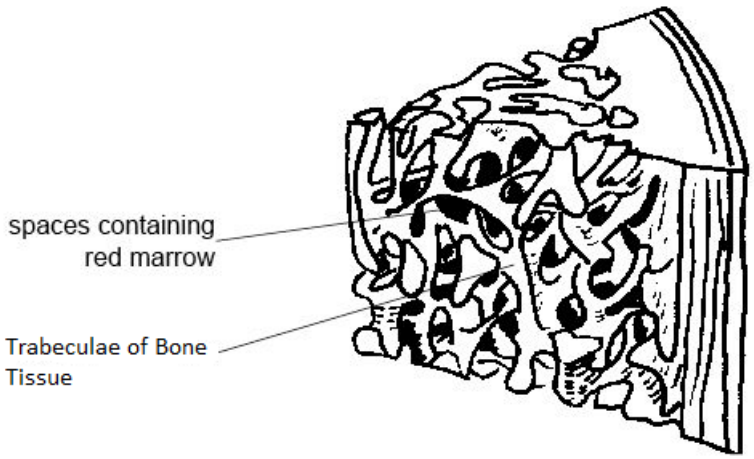


Figure 2.4: Trabeculae [11].

The trabeculae might look randomly arranged since it is hard to see a clear pattern of how the the cell tissue is structured, but it is not. It is constructed and ordered by our body to support the areas which experience the highest stresses. The trabeculae can even grow and change shape and direction in order to give better support to our body, depending on the stresses the body is subjected to. Cancellous bone also contains red bone marrow which fills up the spaces between the trabeculae [12]. Inside the trabecule, there are three types of cells that cooperate to keep the bone strong and healthy: osteoblasts, osteocytes, and osteoclasts. The osteocytes senses when the bone is damaged or subjected to stress. The osteoblasts creates new bone tissue and the osteoclast destroys old or damaged bone in order to make room for the osteoblasts. The procedure of remolding bone tissue by removing old bone tissue and replacing it with new bone tissue is an ongoing and carefully regulated process. The trabeculae need to provide support for the bone without being too thick and dense since it would make the bone unnecessarily heavy and reduce the space for red bone marrow. If it would be too thin however it would make the bone more easily fractured [12].

2.3.2 Teeth

Human teeth are, like most other vertebrate teeth composed of dentin which is capped by a thin layer of enamel, see Figure 2.5. Fully matured enamel consists mainly of mineral (>95% per volume) in form of bundles of highly elongated crystals. This makes the enamel the hardest material in the body, but also brittle. Dentin

contains approximately of 50% crystals, 20% water, and 30% organic matrix per volume. This composition makes it softer, but tougher compared to the enamel covering it [13]. The enamel and the dentin is subjected to cyclic mechanical loading, thermal and hydration stresses as we eat and use our teeth in daily life. The enamel cap transfers the load from its cap and distributes it into the dentin, without having a fracture. It is however the elastic properties of the enamel and the dentin that are important for normal tooth function rather than fracture properties, due to the fatigue damage is always an extreme result of load and normally related to extensive wear [13].

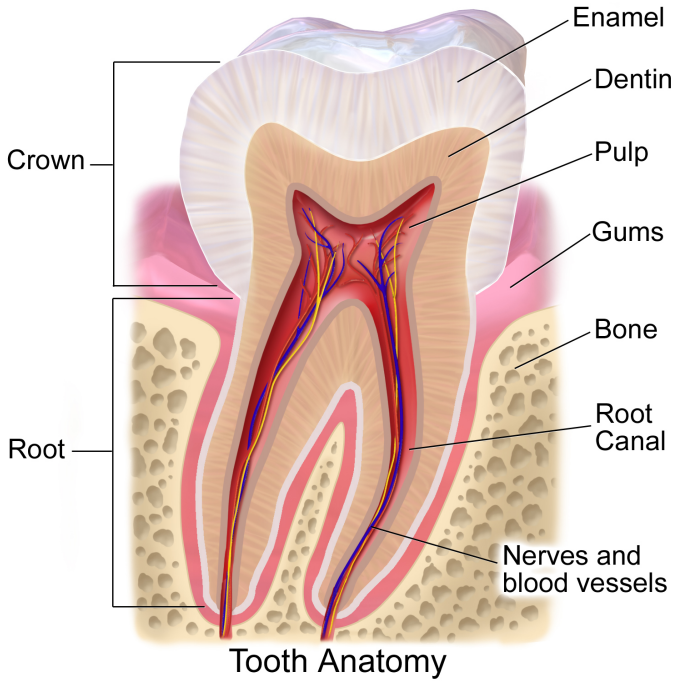


Figure 2.5: The anatomy of a human tooth [14].

2.3.3 Tree

There are two types of wood in trees, softwood and hardwood. The distinction between softwood and hardwood depends on the plants reproduction and its seeds. Hardwood are angiosperms, which means that its seeds have some sort of covering, like fruits or a hard shell like an oaks acorn, Figure 2.6. Softwood trees are gymnosperms, there seeds have no covering and fall to the ground as it is. For example, pine trees are categorized as a softwood tree, where its seeds are spread out in the wind as they are mature.
The structures in hardwood are considered to be more advanced compared to softwood. Hardwood have four main cell types: fibers, vessels, tracheids and parenchyma cells. The fibers are from a mechanical perspective most interesting since they provide the mechanical support and strength to the tree [15]. The three major polymers in wood are cellulose, hemicelluloses, and lignin. Cellulose is the main structural component of the cell wall and the most common macromolecule on earth. The cell wall provides the cell with stability, stiffness, tensile strength and protection from

mechanical stresses [16].

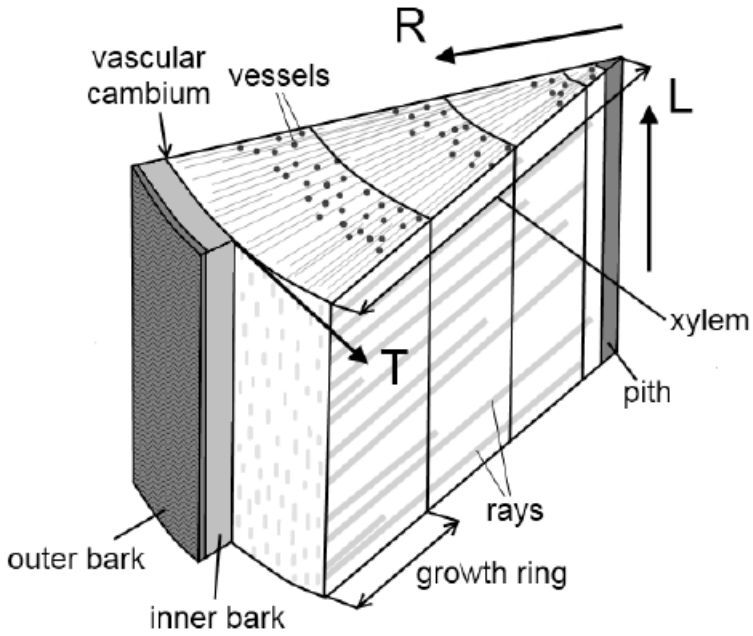


Figure 2.6: Cross-section of a sector of hardwood showing its different layers. **R**, **L** and **T** denotes the radial, longitudinal and tangential direction [15].

On nano-scale cellulose chains are bonded with each other through hydrogen bonds, forming flat sheets. The sheets are in turn stacked on each other forming bundles which is held together by van der Waals forces. These bundels, called fibrils, are orientated different depending its location in the cell wall. In the thicker secondary layer the fibrils are organized in a parallel manner, orientated with a certain angle towards the fibre axis called the microfibril angle. The angle is highly affecting the mechanical properties of the wood. Due to its diverse structure, the mechanical behavior of wood depends highly on the direction and which type of load that is applied [15]. The structure is effective in axial compression which makes the tree withstand its own weight due to gravity and also flexibility in order to be able to bend under windy conditions or external loading such as ice, snow or fruit loading [17].

2.4 Energy absorption fundamentals

Even though energy absorbing structures should suit the particular purpose and circumstances of which they are to work, the aim is to dissipate kinetic energy in a controlled manner or at a predetermined rate. There are fundamental principles that are generally valid for these structures, presented below [18].

Irreversible Energy Conversion

The energy conversion by a structure should be irreversible and convert the input kinetic energy into inelastic energy, such as plastic deformation, rather than storing it in an elastic manner. Meaning, the stresses in the structure should exceed the yield strength of the material.

Restricted and Constant Reactive Force

Ideally the reaction force should remain constant during the deformation process of the energy absorbing structure. The peak reaction force should be kept below a threshold. This peak force correlates to the deceleration and the threshold should be set to a value above which would cause damage or injury. The standards for a bicycle helmet is a deceleration of $300g$ [19].

Long Stroke

In order to keep the reaction force constant and below the threshold, the structure must have a sufficiently long deformation zone. The work done by the force is equal to it's magnitude times the displacement, meaning in order to decrease the force, the displacement needs to increase, see Equation 2.1.

$$W = Fd \tag{2.1}$$

where W is the work, F is the force and d is the deformation. To decelerate uniformly from speed v to 0 m/s requires a distance d, Equation 2.2;

$$d = \frac{vt}{2} \tag{2.2}$$

where t is the time. This distance is what the force acts over to dissipate the kinetic energy. The relation also describes that distance can be "bought" with time. The longer time the force acts, the gentler the arresting force required resulting in a lower risk of injury.

Stable and Repeatable Deformation Mode

Since the loads acting on the structure are varying and uncertain the deformation mode and energy absorption capacity of the design need to be stable and repeatable. This is to ensure reliability of the structure during its service. Examples of uncertainties for the impact could be magnitude, direction and distribution.

2.5 Numerical simulation of energy absorption

There are three main methods to simulate a drop test in Ansys; response spectrum, implicit and explicit [20].

Response Spectrum

Response spectrum assumes the impact to be a half sine loading with a hand calculated time duration expressed as a harmonic frequency. This method requires the model to be completely linear since it is a mode superposition method. The response spectrum solves significantly faster than the transient approaches and uses fewer resources.

Implicit

The implicit method obtains a solution using a series of linear approximations and small iterative time steps are required to achieve convergence. The implicit method is good for drop simulation with long time durations (seconds to minutes) and no or moderate non-linearities. The solution is dependent on current and previous time step and resolves nonlinearities with standard Newton-Raphson iteration approach. The method can handle moderate nonlinearities such as most contact, moderate nonlinear materials and moderate distortion and strain. It uses 2^{nd} order solid elements, hence no hourglass energy issues.

Explicit

The explicit method uses uncoupled equations that can be solved directly (explicit). This method requires tiny time steps that are solved once and no inversion of the stiffness matrix is required. The explicit method is good for problems with short time transients and extreme nonlinearities. This includes extremely large distortions and deformations, material failure and nonlinear materials. The solution depends only on previous time step and requires small time steps (μs) and is limited to problems with duration in milliseconds or less. It uses 1^{st} order elements and need finer mesh to achieve the same accuracy as the implicit model. Ansys tools for handeling explicit dynamics are Ansys Explicit/STR, Ansys/LS-DYNA and Ansys Autodyn.

2.6 Failure criteria for structure - General buckling

Even though buckling on a beam in a structure sometimes does not damage the structure, it must still be avoided since the buckled beam may cause the structure to lose its capability to fulfill its purpose. The actual buckling load may be the final load bearing capacity since the beam in its buckled shape may not sustain any additional load, causing the structure to failure [21]. Therefore, if buckling occur in a layer, the layer will be considered as expired.

3

Theory

3.1 Energy Absorption

In engineering, the evaluation of the energy absorption capacity is defined as the integration of the stress-strain curves, Equation 3.1.

$$E_a = \int_0^{\epsilon_0} \sigma d\epsilon \tag{3.1}$$

E_a is the energy absorption capacity per unit mass, σ is the stress and ϵ_0 is the strain. In order to design an energy absorbing structure, it needs to sustain intense impact loads resulting in deformation and failure involving large geometrical changes, strain-hardening effects, strain-rate effects and different deformation modes like bending and stretching. Because of this, most energy-absorbers are made of ductile materials like low carbon steel, aluminum alloys, polymer foams and fibre-reinforced plastics.

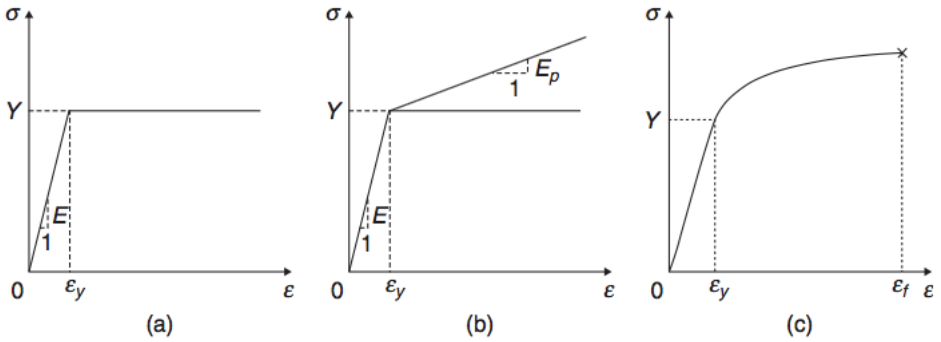


Figure 3.1: Idealised stress-strain curves: (a) elastic, perfectly plastic, (b) elastic, linear hardening and (c) elastic, power hardening [18].

The stresses corresponding to Figure 3.1 relates to the strain ϵ as:

$$\sigma = \begin{cases} E\epsilon & \text{for} \quad \epsilon \leq \epsilon_y = Y/E \\ Y & \text{for} \quad \epsilon_y \leq \epsilon < \epsilon_f \end{cases} \tag{3.2}$$

$$\sigma = \begin{cases} E\epsilon & \text{for} \quad \epsilon \leq \epsilon_y = Y/E \\ Y + E_p(\epsilon - \epsilon_y) & \text{for} \quad \epsilon_y \leq \epsilon < \epsilon_f \end{cases} \tag{3.3}$$

$$\sigma = \begin{cases} E\epsilon & \text{for} \quad \epsilon \leq \epsilon_y = Y/E \\ Y + K(\epsilon - \epsilon_y)^q & \text{for} \quad \epsilon_y \leq \epsilon < \epsilon_f \end{cases} \tag{3.4}$$

where ϵ_y is the yield strain, E_p is the hardening modulus, K and q area material constants determined experimentally [18].

3.2 Finite Element Method

3.2.1 The FEM problem

In order to solve the differential equations for the beam elements, a numerical approach will be applied. The following steps will be carried out in order to solve the problem [22]:

1. Establish the strong formulation of the problem.
2. Obtain the weak formulation by reformulating the strong formulation.
3. Choose approximations for the unknown function.
4. Choose the weight functions according to Galerkin method.
5. Derive element stiffness matrix and element force vector.
6. Solve global system of equations, i.e. the displacements.

Differential equations for Bernoulli's beam theory - strong formulation

Consider an arbitrary structure constructed by n beam elements in the global coordinate system (X, Y, Z). Each element in the structure is considered as Euler-Bernoulli beam with two nodes. Each beam element has 6 degrees of freedom in each node, 3 deformation components (w_x, w_y, w_z) in each coordinate axis direction and 3 rotation components $(\theta_x, \theta_y, \theta_z)$ around each coordinate axis. Now consider an arbitrary beam element with the local coordinate system (x, y, z). The beam is cylindrical with length L and starting as a general case, the beam is subjected to a distributed load $\boldsymbol{q} = \boldsymbol{q}(x)$ and a distributed moment load vector $\boldsymbol{m} = \boldsymbol{m}(x)$. The external loads give rise to an internal force vector $\boldsymbol{F} = \boldsymbol{F}(x)$ and an internal moment vector $\boldsymbol{M} = \boldsymbol{M}(x)$, see Figure 3.2. In order to find the differential equations for the beam element, the procedure outlined in [23] will be adapted. The vectors on component form in the local coordinate system is put as follows:

$$\boldsymbol{F} = \begin{bmatrix} N \\ Q_y \\ Q_z \end{bmatrix}, \quad \boldsymbol{M} = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix}$$

$$\boldsymbol{m} = \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}, \quad \boldsymbol{q} = \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} \tag{3.5}$$

The force N represents the axial force and the components Q_y and Q_z is the shear force in y- and z-directions respectively. The components M_y and M_z denotes the bending moments and the axial component M_x denotes the torsional moment. Force and moment equilibrium of a indefinitely small beam element gives:

$$\begin{aligned} -\boldsymbol{F} + \boldsymbol{F} + d\boldsymbol{F} + \boldsymbol{q}dx = 0 \\ \implies \frac{d\boldsymbol{F}}{dx} + \boldsymbol{q} = \boldsymbol{0} \end{aligned} \tag{3.6}$$

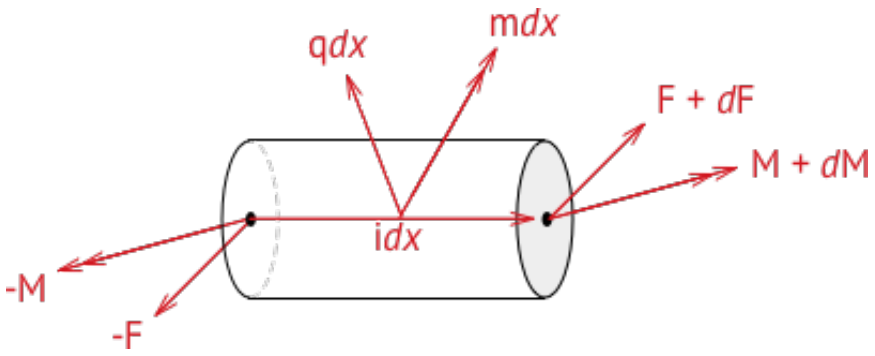


Figure 3.2: An infinitely small beam element.

$$-\boldsymbol{M} + \boldsymbol{M} + dM + \boldsymbol{i} \times (\boldsymbol{F} + d\boldsymbol{F})dx + \boldsymbol{m}dx = \boldsymbol{0} \tag{3.7}$$

By neglecting second order terms yields:

$$\frac{d\boldsymbol{M}}{dx} + \boldsymbol{i} \times \boldsymbol{F} + \boldsymbol{m} = \boldsymbol{0} \tag{3.8}$$

Now, by using

$$\boldsymbol{i} \times \boldsymbol{F} = \boldsymbol{i} \times (N_i + Q_y\boldsymbol{j} + Q_z\boldsymbol{k}) = N\boldsymbol{i} \times \boldsymbol{i} + Q_y\boldsymbol{i} \times jQ_z\boldsymbol{i} \times \boldsymbol{k} = 0\boldsymbol{i} - Q_z + Q_y\boldsymbol{k}$$

the following components relations can be expressed from Equation 3.6 and 3.8:

$$\frac{dN}{dx} + q_x = 0, \quad \frac{dQ_y}{dx} + q_y = 0, \quad \frac{dQ_z}{dx} + q_z = 0, \tag{3.9}$$

$$\frac{dM_x}{dx} + m_x = 0, \quad \frac{dM_y}{dx} - Q_z + m_y = 0, \quad \frac{dM_z}{dx} + Q_y + m_z = 0 \tag{3.10}$$

Using kinematic relations for a Bernoulli-Euler beam, which states that rotated cross-section is always orthogonal to the deformed beam axis, the rotation θ and curvature κ can be expressed as:

$$\theta_y = -\frac{dw_z}{dx}, \quad \theta_z = \frac{dw_y}{dx} \tag{3.11}$$

$$\kappa_y = -\frac{d\theta_y}{dx} = \frac{d^2w_z}{dx^2}, \quad \kappa_z = \frac{d\theta_z}{dx} = \frac{d^2w_y}{dx^2} \tag{3.12}$$

The beam axis is considered to be located in the neutral axis, where there are no longitudinal stresses or strains. This results in the axial and bending problem is uncoupled and can be separately examined. Assuming homogeneous material, the moment is:

$$M = -EI\kappa \tag{3.13}$$

This, together by taking the first derivative of the bending equations in 3.10 and use the expressions in Equation 3.9 yields the differential equations for bending:

$$\frac{d^2}{dx^2}\left(EI_z\frac{d^2w_y}{dx^2}\right) - q_y + \frac{dm_z}{dx} = 0$$

$$\frac{d^2}{dx^2}\left(EI_y\frac{d^2w_z}{dx^2}\right) - q_z + \frac{dm_y}{dx} = 0 \tag{3.14}$$

The differential equation for the axial deformation can be expressed by rewriting the axial component with the the normal force N in Equation 3.9. In terms of normal stress $N = \sigma A$ and using Hookes law ($\sigma = E\epsilon$) and the kinematic relation $\epsilon = dw_x/dx$, Equation 3.9 leads to the sought differential equation for axial deformation:

$$\frac{d}{dx}\left(EA\frac{dw_x}{dx}\right) + q_x = 0$$

Since the beam is three dimensional, the twist of the beam must be taken into consideration as well. The torsion is assumed to be homogeneous, i.e. M_x, $\frac{d\theta_x}{dx}$ and the warping of the cross sections is constant along the beam (also known as St. Venant torsion). The beam material is assumed to be homogeneous and isotropic linear elastic with shear modulus G. In homogeneous torsion, warping does not induce normal strains i.e. $\epsilon_{xx} = \frac{du_x}{dx} = 0$ which in turn leads to no normal stress as well (Hookes law: $\sigma_{xx} = E\epsilon_{xx}$). Thus, only shear stresses are present in the cross section. The following relation can then be expressed by the linearity assumption that the torsional moment depends linearly on the twist gradient and the shear stresses depends linearly on the shear modulus [23]:

$$M_x = GK\frac{d\theta_x}{dx} \tag{3.15}$$

where G is the shear modulus and K is St. Venant torsion constant. Taking the first derivative of Equation 3.15 and insert it into Equation 3.10 yields the sought differential equation for torsion:

$$GK\frac{d^2\theta_x}{dx^2} + m_x = 0 \tag{3.16}$$

Summarizing, the following differential equations have been expressed for the beam, i.e. the strong formulation:

$$\frac{d}{dx}\left(EA\frac{dw_x}{dx}\right) + q_x = 0 \tag{3.17}$$

$$\frac{d^2}{dx^2}\left(EI_y\frac{d^2w_z}{dx^2}\right) - q_z - \frac{dm_y}{dx} = 0 \tag{3.18}$$

$$\frac{d^2}{dx^2}\left(EI_z\frac{d^2w_y}{dx^2}\right) - q_y + \frac{dm_z}{dx} = 0 \tag{3.19}$$

$$GK\frac{d^2\theta_x}{dx^2} + m_x = 0 \tag{3.20}$$

The weak formulation

In order to find the weak form of the differential equations, an arbitrary function $v(x)$ is multiplied with each one of the differential equations and integrated over the pertinent region. Starting with Equation 3.17:

$$\int_a^b v\Big(\frac{d}{dx}\Big(AE\frac{dw_x}{dx}\Big) + q_x\Big)dx = 0 \qquad a \leq x \leq b \tag{3.21}$$

By integrating by parts, the weak formulation of axial deformation is obtained:

$$\int_a^b \frac{dv}{dx} AE\frac{dw_x}{dx}dx = \Big[vAE\frac{dw_x}{dx}\Big]_a^b + \int_a^b vq_x dx \tag{3.22}$$

The weak form of bending in the xz-plane is obtained in the same manner by first multiplying an arbitrary function $v(x)$ to Equation 3.18 and integrate over the pertinent region:

$$\int_a^b v\frac{d^2}{dx^2}\Big(EI_y\frac{d^2w_z}{dx^2}\Big)dx - \int_a^b vq_z dx - \int_a^b v\frac{dm_y}{dx}dx = 0 \tag{3.23}$$

Integrating 3.23 by parts twice:

$$-\Big[vV_z\Big]_a^b - \int_a^b \frac{dv}{dx}\frac{d}{dx}\Big(EI_y\frac{d^2w_z}{dx^2}\Big)dx - \int_a^b vq_z dx - \int_a^b v\frac{dm_y}{dx}dx = 0 \tag{3.24}$$

$$-\Big[vV_z\Big]_a^b + \Big[\frac{dv}{dx}M_y\Big]_a^b + \int_a^b \frac{d^2v}{dx^2}EI_y\frac{d^2w_z}{dx^2} - \int_a^b vq_z dx - \int_a^b v\frac{dm_y}{dx}dx = 0 \tag{3.25}$$

Rearranging the boundary terms and distributed load to RHS:

$$\implies \int_a^b \frac{d^2v}{dx^2}EI_y\frac{d^2w_z}{dx^2}dx = \Big[vV_z\Big]_a^b - \Big[\frac{dv}{dx}M_z\Big]_a^b + \int_a^b vq_z dx - \int_a^b v\frac{dm_y}{dx}dx \tag{3.26}$$

where

$$V_z = -\frac{d}{dx}\Big(EI_y\frac{d^2w_z}{dx^2}\Big), \quad M_y = -EI_y\frac{d^2w_z}{dx^2}$$

Adopting the same procedure for Equation 3.19 gives the weak formulation of bending in xy-plane:

$$\implies \int_a^b \frac{d^2v}{dx^2}EI_z\frac{d^2w_y}{dx^2}dx = \Big[vV_y\Big]_a^b - \Big[\frac{dv}{dx}M_z\Big]_a^b + \int_a^b vq_y dx - \int_a^b v\frac{dm_y}{dx}dx \tag{3.27}$$

where

$$V_y = -\frac{d}{dx}\Big(EI_z\frac{d^2w_y}{dx^2}\Big), \quad M_z = -EI_y\frac{d^2w_y}{dx^2}$$

Lastly, the weak formulation of torsion is derived from Equation 3.20 by using the same procedure to:

$$GK\int_a^b \frac{dv}{dx}\frac{d\theta_x}{dx}dx = \Big[vGK\frac{d\theta_x}{dx}\Big]_a^b + \int_a^b vm_x dx \tag{3.28}$$

FE-formulation

From the weak formulation of the equilibrium equations, the FE-formulation is derived [22]. Since the deflection w is the unknown function, the approximation for the deflection w of one element can be generally written as:

$$\boldsymbol{w} = \boldsymbol{N}\boldsymbol{a} \tag{3.29}$$

where

$$\boldsymbol{N} = [N_1 \ N_2 \ \dots \ N_n], \quad \boldsymbol{a} = [u_1 \ u_2 \ \dots \ u_n]^T \tag{3.30}$$

n is the number of unknown for the entire beam. Starting with the bending equations, 3.26 and 3.27. From Equation 3.29 it follows that:

$$\frac{d^2w}{dx^2} = \boldsymbol{B}\boldsymbol{a}, \quad \text{where} \quad \boldsymbol{B} = \frac{d^2\boldsymbol{N}}{dx^2} \tag{3.31}$$

The arbitrary weight functions v is now chosen according to Galerkin:

$$v = \boldsymbol{N}\boldsymbol{c} \tag{3.32}$$

It is concluded that the parameters given by $\boldsymbol{c}$ are arbitrary since the weight functions are arbitrary. The weight function can be rewritten to:

$$v = \boldsymbol{c}^T\boldsymbol{N}^T$$

$$\frac{dv}{dx} = \boldsymbol{c}^T\frac{d\boldsymbol{N}^T}{dx}, \qquad \frac{d^2v}{dx^2} = \boldsymbol{c}^T\boldsymbol{B}^T \tag{3.33}$$

Inserting variables from Equation 3.33 into the weak formulation of bending in $xz-$plane, Equation 3.26, yields:

$$\boldsymbol{c}^T\Big(\int_a^b \boldsymbol{B}^T EI_y \boldsymbol{B}dx\boldsymbol{a}\Big) = \boldsymbol{c}^T\Big(\Big[\boldsymbol{N}^T V_z\Big]_a^b - \Big[\frac{d\boldsymbol{N}^T}{dx}M_y\Big]_a^b + \int_a^b \boldsymbol{N}^T q_z dx + \int_a^b \boldsymbol{N}^T\frac{dm_y}{dx}dx\Big) \tag{3.34}$$

Since $\boldsymbol{c}^T$ is arbitrary chosen it is concluded that:

$$\int_a^b \boldsymbol{B}^T EI_y \boldsymbol{B}dx\boldsymbol{a} = \Big[\boldsymbol{N}^T V_z\Big]_a^b - \Big[\frac{d\boldsymbol{N}^T}{dx}M_y\Big]_a^b + \int_a^b \boldsymbol{N}^T q_z dx + \int_a^b \boldsymbol{N}^T\frac{dm_y}{dx}dx \tag{3.35}$$

which is the sought FE-formulation for bending in xz-plane. Adopting the same procedure to Equation 3.27 yields the FE-formulation for bending in xy-plane to:

$$\int_a^b \boldsymbol{B}^T EI_z \boldsymbol{B}dx\boldsymbol{a} = \Big[\boldsymbol{N}^T V_y\Big]_a^b - \Big[\frac{d\boldsymbol{N}^T}{dx}M_z\Big]_0^L + \int_a^b \boldsymbol{N}^T q_y dx - \int_a^b \boldsymbol{N}^T\frac{dm_z}{dx}dx \tag{3.36}$$

It is desired to write the FE-formulation in compact form $\boldsymbol{K}\boldsymbol{a} = \boldsymbol{f}$, which gives in xz-plane:

$$\boldsymbol{K} = \int_a^b \boldsymbol{B}^T EI_y \boldsymbol{B}dx$$

$$\boldsymbol{f} = \boldsymbol{f}_b + \boldsymbol{f}_l$$

where

$$\boldsymbol{f}_b = [\boldsymbol{N}^T V_z]_b^a - \left[\frac{d\boldsymbol{N}^T}{dx} M_y\right]_b^a \quad \text{and} \quad \boldsymbol{f}_l = \int_a^b \boldsymbol{N}^T q_z dx + \int_a^b \boldsymbol{N}^T \frac{dm_y}{dx} dx \tag{3.37}$$

and in xy-plane:

$$\boldsymbol{K} = \int_a^b \boldsymbol{B}^T E I_z \boldsymbol{B} dx$$
$$\boldsymbol{f} = \boldsymbol{f}_b + \boldsymbol{f}_l$$

where

$$\boldsymbol{f}_b = [\boldsymbol{N}^T V_y]_b^a - \left[\frac{d\boldsymbol{N}^T}{dx}\right]_b^a \quad \text{and} \quad \boldsymbol{f}_l = \int_a^b \boldsymbol{N}^T q_y dx - \int_a^b \boldsymbol{N}^T \frac{dm_z}{dx} dx \tag{3.38}$$

$\boldsymbol{K}$ is the stiffness matrix, $\boldsymbol{f}_l$ the load vector and $\boldsymbol{f}_b$ is the boundary vector. The FE-formulation of axial deformation is obtained in the same manner by defining $\boldsymbol{B}$ and chose weight functions according to Galerkin. The FE-formulation for axial deformation, starting from Equation 3.22, yields:

$$\int_a^b \boldsymbol{B}^T AE \boldsymbol{B} dx \boldsymbol{a} = \left[\boldsymbol{N}^T N_x\right]_a^b + \int_a^b \boldsymbol{N}^T q_x dx$$

where

$$N_x = AE \frac{du_x}{dx}, \quad \boldsymbol{B} = \frac{d\boldsymbol{N}_x}{dx} \tag{3.39}$$

In compact form:

$$\boldsymbol{K} = \int_a^b \boldsymbol{B}^T AE \boldsymbol{B} dx, \quad \boldsymbol{B} = \frac{d\boldsymbol{N}}{dx}$$
$$\boldsymbol{f} = \left[\boldsymbol{N}^T N_x\right]_a^b + \int_a^b \boldsymbol{N}^T q_x dx \tag{3.40}$$

The FE-formulation of torsion is obtained from Equation 3.28 in the same manner to:

$$\boldsymbol{K} = \int_a^b \boldsymbol{B}^T GK \boldsymbol{B} dx, \quad \boldsymbol{B} = \frac{d\boldsymbol{N}}{dx}$$
$$\boldsymbol{f} = \left[\boldsymbol{N}^T M_x\right]_a^b + \int_a^b \boldsymbol{N}^T m_x dx \tag{3.41}$$

where

$$M_x = GK \frac{d\theta_x}{dx} \tag{3.42}$$

Evaluation of element stiffness matrix

Since the 3D-beam element have 6 degrees of freedom in each of its two nodes the total unknowns for the element is $n_e = 12$.

$$\boldsymbol{a}^e = [u_1 \ u_2 \ u_3 \ u_4 \ u_5 \ u_6 \ u_7 \ u_8 \ u_9 \ u_{10} \ u_{11} \ u_{12} \ ]^T$$

$$\boldsymbol{N}^e = [N_1 \ N_2 \ N_3 \ N_4 \ N_5 \ N_6 \ N_7 \ N_8 \ N_9 \ N_{10} \ N_{11} \ N_{12}] \tag{3.43}$$

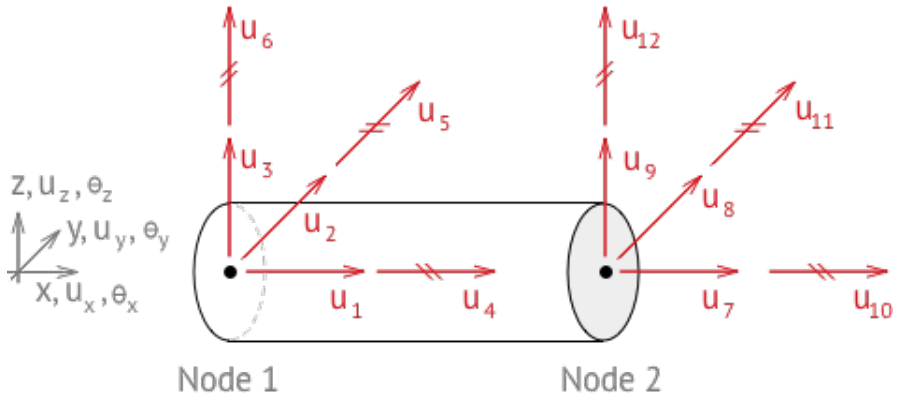


Figure 3.3: A beam element with one node at each end. In total, the beam has 12 degrees of freedom represented by u_{1-12}.

Where u represents each degree of freedom (see Figure 3.3) and N is the chosen shape functions. Starting with bending equation in xz-plane, Equation 3.37, it includes two degrees of freedom, u_3, u_5 and u_9, u_{11} for the first and second node respectively. Considering a beam element with length $0 \leq x \leq L$, the contribution to the element stiffness matrix $\boldsymbol{K}^e$ from bending in xz-plane can be calculated from:

$$\boldsymbol{K}^e = \int_0^L \boldsymbol{B}^{eT} EI_y \boldsymbol{B}^e dx, \quad \boldsymbol{B}^e = \frac{d^2 \boldsymbol{N}^e}{dx^2} \tag{3.44}$$

The corresponding shape functions are calculated by adopting the $\boldsymbol{C}$-matrix method [22].

$$N_3 = 1 - 3\frac{x^2}{L^2} + 2\frac{x^3}{L^3}, \quad N_5 = x\left(1 - 2\frac{x}{L} + \frac{x^2}{L^2}\right)$$

$$N_9 = \frac{x^2}{L^2}\left(3 - 2\frac{x}{L}\right), \quad N_{11} = \frac{x^2}{L^2}\left(\frac{x}{L} - 1\right) \tag{3.45}$$

By using the definition of $\boldsymbol{B}^e$ from Equation 3.44 and take the second derivative of the chosen shape functions in Equation 3.48, one obtains:

$$\boldsymbol{K}^e = EI_y \int_0^L \begin{bmatrix} B_3^e B_3^e & B_3^e B_5^e & B_3^e B_9^e & B_3^e B_{11}^e \\ B_5^e B_3^e & B_5^e B_5^e & B_5^e B_9^e & B_5^e B_{11}^e \\ B_9^e B_3^e & B_9^e B_5^e & B_9^e B_9^e & B_9^e B_{11}^e \\ B_{11}^e B_3^e & B_{11}^e B_5^e & B_{11}^e B_9^e & B_{11}^e B_{11}^e \end{bmatrix}, \quad \boldsymbol{B}^{eT} = \begin{bmatrix} 0 \\ 0 \\ \frac{12x}{L^3} - \frac{6}{L^2} \\ 0 \\ \frac{6x}{L^2} - \frac{4}{L} \\ 0 \\ 0 \\ 0 \\ \frac{6}{L^2} - \frac{12x}{L^3} \\ 0 \\ \frac{6x}{L^2} - \frac{2}{L} \\ 0 \end{bmatrix} \tag{3.46}$$

Solving the integral in Equation 3.46 gives the following contributions to the element stiffness matrix:

$$K^e(3,3) = EI_y\frac{12}{L^3}, \quad K^e(3,5) = EI_y\frac{6}{L^2}, \quad K^e(3,9) = -EI_y\frac{12}{L^3}$$

$$K^e(3,11) = EI_y\frac{6}{L^2}, \quad K^e(5,3) = EI_y\frac{6}{L^2}, \quad K^e(5,5) = EI_y\frac{4}{L}$$

$$K^e(5,9) = -EI_y\frac{6}{L^2}, \quad K^e(5,11) = EI_y\frac{2}{L}, \quad K^e(9,3) = -EI_y\frac{12}{L^3},$$

$$K^e(9,5) = -EI_y\frac{6}{L^2}, \quad K^e(9,9) = EI_y\frac{12}{L^3}, \quad K^e(9,11) = -EI_y\frac{6}{L^2}$$

$$K^e(11,3) = EI_y\frac{6}{L^2}, \quad K^e(11,5) = EI_y\frac{2}{L}, \quad K^e(11,9) = -EI_y\frac{6}{L^2},$$

$$K^e(11,11) = EI_y\frac{4}{L} \tag{3.47}$$

The contribution to $\boldsymbol{K}^e$ from bending in xy-plane is obtained in the same manner as with the bending in xz-plane by calculating shape functions using $\boldsymbol{C}$-matrix method and the FE formulation from Equation 3.37. It is noted that the shape functions for bending in xy-plane (N_2, N_6, N_8, N_{12}) are identical to the shape functions in xz-plane, i.e.

$$N_2 = N_3 = 1 - 3\frac{x^2}{L^2} + 2\frac{x^3}{L^3}, \quad N_6 = N_5 = x\Big(-1 + 2\frac{x}{L} - \frac{x^2}{L^2}\Big)$$

$$N_8 = N_9 = \frac{x^2}{L^2}\Big(3 - 2\frac{x}{L}\Big), \quad N_{12} = N_{11} = \frac{x^2}{L^2}\Big(1 - \frac{x}{L}\Big) \tag{3.48}$$

thus:

$$\boldsymbol{K}^e = EI_z\int_0^L \begin{bmatrix} B_2^eB_2^e & B_2^eB_6^e & B_2^eB_8^e & B_2^eB_{12}^e \\ B_6^eB_2^e & B_6^eB_6^e & B_6^eB_8^e & B_6^eB_{12}^e \\ B_8^eB_2^e & B_8^eB_6^e & B_8^eB_8^e & B_8^eB_{12}^e \\ B_{12}^eB_2^e & B_{12}^eB_6^e & B_{12}^eB_8^e & B_{12}^eB_{12}^e \end{bmatrix}, \quad \boldsymbol{B}^{eT} = \begin{bmatrix} 0 \\ \frac{12x}{L^3} - \frac{6}{L^2} \\ 0 \\ 0 \\ 0 \\ \frac{6x}{L^2} - \frac{4}{L} \\ 0 \\ \frac{6}{L^2} - \frac{12x}{L^3} \\ 0 \\ 0 \\ 0 \\ \frac{6x}{L^2} - \frac{2}{L} \end{bmatrix} \tag{3.49}$$

Solving the integral in Equation 3.49 gives the contribution from bending in xy-plane:

$$K^e(2,2) = EI_z\frac{12}{L^3}, \quad K^e(2,6) = EI_z\frac{6}{L^2}, \quad K^e(2,8) = -EI_z\frac{12}{L^3}$$

$$K^e(2,12) = EI_z\frac{6}{L^2}, \quad K^e(6,2) = EI_z\frac{6}{L^2}, \quad K^e(6,6) = EI_z\frac{4}{L}$$
$$K^e(6,8) = -EI_z\frac{6}{L^2}, \quad K^e(6,12) = EI_z\frac{2}{L}, \quad K^e(8,2) = -EI_z\frac{12}{L^3},$$
$$K^e(8,6) = -EI_z\frac{6}{L^2}, \quad K^e(8,8) = EI_z\frac{12}{L^3}, \quad K^e(8,12) = -EI_z\frac{6}{L^2}$$
$$K^e(12,2) = EI_z\frac{6}{L^2}, \quad K^e(12,6) = EI_z\frac{2}{L}, \quad K^e(12,8) = -EI_z\frac{6}{L^2},$$
$$K^e(12,12) = EI_z\frac{4}{L} \tag{3.50}$$

The contribution to the stiffness matrix from axial deformation is calculated from Equation 3.40:

$$\boldsymbol{K}^e = \int_0^L \boldsymbol{B}^e AE\boldsymbol{B}^{eT}dx, \quad \boldsymbol{B}^e = \frac{d\boldsymbol{N}^e}{dx} \tag{3.51}$$

The corresponding degrees of freedom for the element are u_1 and u_7, see Figure 3.3. Since there are only two degrees of freedom for axial deformation, the shape functions N_1 and N_7 are easily chosen as:

$$N_1 = \frac{1}{L}(L-x), \quad N_7 = \frac{x}{L} \tag{3.52}$$

Taking the first derivative of $N1$ and $N7$ and inserting them into Equation 3.51 yields:

$$\boldsymbol{K}^e = EA\int_0^L \begin{bmatrix} B_1^eB_1^e & B_1^eB_7^e \\ B_7^eB_1^e & B_7^eB_7^e \end{bmatrix}, \quad \boldsymbol{B}^{eT} = \begin{bmatrix} -\frac{1}{L} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{L} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.53}$$

Solving the integral in Equation 3.53, the contribution to $\boldsymbol{K}^e$ from axial deformation is obtained as:

$$K^e(1,1) = \frac{AE}{L}, \quad K^e(1,7) = -\frac{AE}{L}, \quad K^e(7,1) = -\frac{AE}{L}, \quad K^e(7,7) = \frac{AE}{L} \tag{3.54}$$

The contribution to the element stiffness matrix from torsion remains to be derived. The shape functions for torsion is identical to the shape functions in axial deformation i.e.

$$N_1 = N_4 = \frac{1}{L}(L-x), \quad N_7 = N_{10} = \frac{x}{L} \tag{3.55}$$

Inserting the shape functions into Equation 3.41 yields:

$$\boldsymbol{K}^e = GK \int_0^L \begin{bmatrix} B_4^e B_4^e & B_4^e B_{10}^e \\ B_{10}^e B_4^e & B_{10}^e B_{10}^e \end{bmatrix}, \quad \boldsymbol{B}^{eT} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -\frac{1}{L} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{L} \\ 0 \\ 0 \end{bmatrix} \tag{3.56}$$

$$\begin{aligned} K^e(4,4) &= \frac{GK}{L}, \quad K^e(4,10) = -\frac{GK}{L} \\ K^e(10,4) &= -\frac{GK}{L}, \quad K^e(10,10) = \frac{GK}{L} \end{aligned} \tag{3.57}$$

Summarizing the contributions from Equation 3.47, 3.50, 3.54 and 3.57 yields the full element stiffness matrix:

$$\boldsymbol{K}^e = \begin{bmatrix} \frac{AE}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{AE}{L} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} & 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & \frac{6EI_z}{L^2} \\ 0 & 0 & \frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^2} & 0 & 0 & 0 & -\frac{12EI_y}{L^3} & 0 & \frac{6EI_y}{L^3} & 0 \\ 0 & 0 & 0 & \frac{GK}{L} & 0 & 0 & 0 & 0 & 0 & -\frac{GK}{L} & 0 & 0 \\ 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 & 0 & 0 & -\frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 \\ 0 & \frac{6EI_y}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} \\ -\frac{AE}{L} & 0 & 0 & 0 & 0 & 0 & \frac{AE}{L} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} & 0 & \frac{12EI_z}{L^3} & 0 & 0 & 0 & -\frac{6EI_z}{L^2} \\ 0 & 0 & -\frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 & 0 & 0 & \frac{12EI_y}{L^3} & 0 & -\frac{6EI_y}{L^2} & 0 \\ 0 & 0 & 0 & -\frac{GK}{L} & 0 & 0 & 0 & 0 & 0 & \frac{GK}{L} & 0 & 0 \\ 0 & 0 & \frac{6EI_y}{L^2} & 0 & \frac{2EI_y}{L} & 0 & 0 & 0 & -\frac{6EI_y}{L^2} & 0 & \frac{4EI_y}{L} & 0 \\ 0 & \frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{2EI_z}{L} & 0 & -\frac{6EI_z}{L^2} & 0 & 0 & 0 & \frac{4EI_z}{L} \end{bmatrix}$$

$\boldsymbol{K}^e$ is defined in the elements local coordinate system (x, y, z) and in order to be able to assemble the full stiffness matrix $\boldsymbol{K}$, $\boldsymbol{K}^e$ needs to be transformed and expressed in the global coordinate system (X, Y, Z). The transformation between coordinate systems is done by using a transformation matrix $\boldsymbol{G}$ according to:

$$\boldsymbol{K}^e = \boldsymbol{G}^T \bar{\boldsymbol{K}}^e \boldsymbol{G} \tag{3.58}$$

where $\bar{\boldsymbol{K}}^e$ now denotes the element stiffness matrix in local coordinate system and $\boldsymbol{K}^e$ denotes the element stiffness matrix in global coordinate system [24]. How the element stiffness matrix and transform matrix $\boldsymbol{G}$ is implemented is further discussed in Section 3.2.2.

Evaluation of force vector

The element force vector $\boldsymbol{f}^e$ is defined as the sum of the boundary vector and the load vector according to $\boldsymbol{f}^e = \boldsymbol{f}_b^e + \boldsymbol{f}_l^e$. Since no distributed load is present in the structure i.e. $q_x = q_y = q_z = 0$, there is no contribution from the element load vector to the solution. The standard form is thus reduced to:

$$\boldsymbol{K}^e \boldsymbol{a}^e = \boldsymbol{f}_b^e \tag{3.59}$$

In Figure 3.4, boundaries of an arbitrary structure of beam elements is displayed. The boundary Γ_{sym} simulates neighbouring cell connections. Nodes which lays on Γ_{sym} are ruled by a Dirichlet boundary condition, preventing any movement in $xy-$plane and only allow movement in z-direction. The boundary Γ_g represents the ground, thus nodes present on Γ_g is prevented from any movement in z-direction. Nodes located at the boundary at the top of the structure Γ_f is subjected to a prescribed force. The magnitude of the force is dependent on type of structure and is further explained and calculated in Section 4.3.

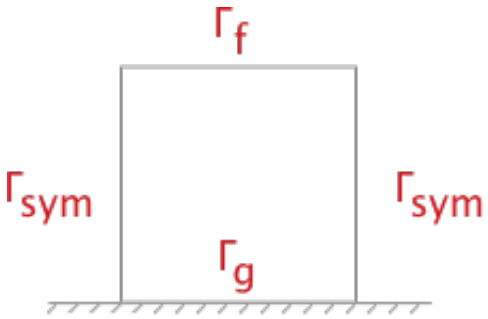


Figure 3.4: A beam element with one node at each end. In total, the beam has 12 degrees of freedom represented by u_{1-12}.

3.2.2 CALFEM

In order to apply the derived formulations of the FEM-problem in Section 3.2.1, CALFEM is used. CALFEM is a Matlab toolbox computer program for finite element applications. The program contains a library of finite element methods handling matrix operations, material, element, system, statement and graphical functions. This thesis uses CALFEM's beam elements to represent each connection link and all beams within the geometry.

The beam3e function

The beam3e function computes the element stiffness matrix for a three dimensional beam element [24]. This provides the global element stiffness matrix K_e for the beam element, Equation 3.60.

$$\boldsymbol{K}^e = \text{beam3e}(e_x, e_y, e_z, e_o, e_p), \quad \begin{cases} e_x = [x_1, x_2] \\ e_y = [y_1, y_2] \\ e_z = [z_1, z_2] \\ e_o = [x_{\bar{z}}, y_{\bar{z}}, z_{\bar{z}}] \\ e_p = [E, G, A, I_{\bar{y}}, I_{\bar{z}}, K_v] \end{cases} \tag{3.60}$$

The input variables supply the element nodal coordinates $(x_1, y_1, ...)$, the direction of the local beam coordinate system $(x_{\bar{z}}, y_{\bar{z}}, z_{\bar{z}})$, see Figure 3.5, and material data (e_p). The material data needed is modulus of elasticity E, shear modulus G, cross sectional area A, moment of inertia with respect to the $\bar{y}$ and $\bar{z}$ axis, $I_{\bar{y}}$ & $I_{\bar{z}}$ and St Venant torsinal stiffness K_v.

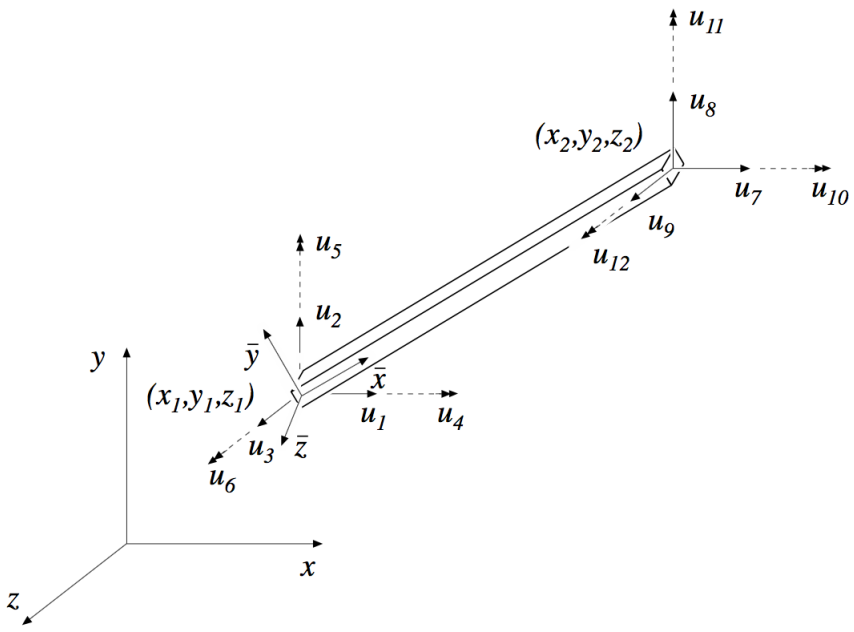


Figure 3.5: The CALFEM beam element in three dimensions displaying degrees of freedom and the local coordinate system $(\bar{x}, \bar{y}, \bar{z})$ [24].

The element stiffness matrix K^e is computed according to Equation 3.61.

$$\boldsymbol{K}^e = \boldsymbol{G}^T \bar{\boldsymbol{K}}^e \boldsymbol{G} \tag{3.61}$$

where

$$\bar{\boldsymbol{K}}^e = \begin{bmatrix} k_1 & 0 & 0 & 0 & 0 & 0 & -k_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{12EI_{\bar{z}}}{L^3} & 0 & 0 & 0 & \frac{6EI_{\bar{z}}}{L^2} & 0 & -\frac{12EI_{\bar{z}}}{L^3} & 0 & 0 & 0 & \frac{6EI_{\bar{z}}}{L^2} \\ 0 & 0 & \frac{12EI_{\bar{y}}}{L^3} & 0 & -\frac{6EI_{\bar{y}}}{L^2} & 0 & 0 & 0 & -\frac{12EI_{\bar{y}}}{L^3} & 0 & -\frac{6EI_{\bar{y}}}{L^2} & 0 \\ 0 & 0 & 0 & k_2 & 0 & 0 & 0 & 0 & 0 & -k_2 & 0 & 0 \\ 0 & 0 & \frac{6EI_{\bar{y}}}{L^2} & 0 & \frac{4EI_{\bar{y}}}{L} & 0 & 0 & 0 & \frac{6EI_{\bar{y}}}{L^2} & 0 & \frac{2EI_{\bar{y}}}{L} & 0 \\ 0 & -\frac{6EI_{\bar{y}}}{L^2} & 0 & 0 & 0 & \frac{4EI_{\bar{z}}}{L} & 0 & -\frac{6EI_{\bar{z}}}{L^2} & 0 & 0 & 0 & \frac{2EI_{\bar{z}}}{L} \\ -k_1 & 0 & 0 & 0 & 0 & 0 & k_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{12EI_{\bar{z}}}{L^3} & 0 & 0 & 0 & -\frac{6EI_{\bar{z}}}{L^2} & 0 & \frac{12EI_{\bar{z}}}{L^3} & 0 & 0 & 0 & -\frac{6EI_{\bar{z}}}{L^2} \\ 0 & 0 & -\frac{12EI_{\bar{y}}}{L^3} & 0 & \frac{6EI_{\bar{y}}}{L^2} & 0 & 0 & 0 & \frac{12EI_{\bar{y}}}{L^3} & 0 & \frac{6EI_{\bar{y}}}{L^2} & 0 \\ 0 & 0 & 0 & -k_2 & 0 & 0 & 0 & 0 & 0 & k_2 & 0 & 0 \\ 0 & 0 & -\frac{6EI_{\bar{y}}}{L^2} & 0 & \frac{2EI_{\bar{y}}}{L} & 0 & 0 & 0 & \frac{6EI_{\bar{y}}}{L^2} & 0 & \frac{4EI_{\bar{y}}}{L} & 0 \\ 0 & \frac{6EI_{\bar{z}}}{L^2} & 0 & 0 & 0 & \frac{2EI_{\bar{z}}}{L} & 0 & -\frac{6EI_{\bar{z}}}{L^2} & 0 & 0 & 0 & \frac{4EI_{\bar{z}}}{L} \end{bmatrix}$$

in which $k_1 = \frac{EA}{L}$ and $k_2 = \frac{GK_v}{L}$, and

$$\boldsymbol{G} = \begin{bmatrix} n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{x}} & n_{y\bar{x}} & n_{z\bar{x}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{y}} & n_{y\bar{y}} & n_{z\bar{y}} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n_{x\bar{z}} & n_{y\bar{z}} & n_{z\bar{z}} \end{bmatrix}$$

in which $n_{x\bar{x}}$ specifies the cosine of the angle between the x axis and $\bar{x}$ axis and so on. The beam element length L is computed from Equation 3.62.

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{3.62}$$

The beam3s function

The beam3s function computes the section forces and displacements, Equation 3.63, in local directions along the three dimensional beam element [24]. Figure 3.6 displays one section element of the beam and the computed forces and displacements.

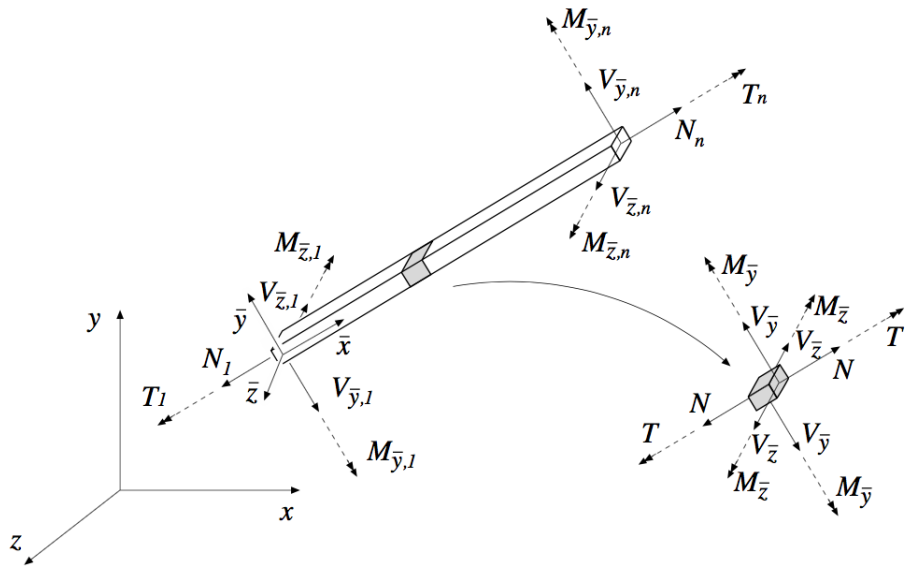


Figure 3.6: The CALFEM beam element in three dimensions displaying degrees of freedom and the local coordinate system $(\bar{x}, \bar{y}, \bar{z})$ [24].

$$e_s = \text{beam3s}(e_x, e_y, e_z, e_o, e_p, e_d), \quad \begin{cases} e_x = [x_1, x_2] \\ e_y = [y_1, y_2] \\ e_z = [z_1, z_2] \\ e_o = [x_{\bar{z}}, y_{\bar{z}}, z_{\bar{z}}] \\ e_p = [E, G, A, I_{\bar{y}}, I_{\bar{z}}, K_v] \end{cases} \tag{3.63}$$

The element displacements, stored in e_d obtained by the *extract* function. The output variable consists of column matrices that contain the section forces, displacements and the evaluation points on the local $\bar{x}$-axis as $e_s = [N, V_{\bar{y}}, V_{\bar{z}}, T, M_{\bar{y}}, M_{\bar{z}}]$ or

$$e_s = \begin{bmatrix} N_1 & V_{\bar{y}1} & V_{\bar{z}1} & T_1 & M_{\bar{y}1} & M_{\bar{z}1} \\ N_2 & V_{\bar{y}2} & V_{\bar{z}2} & T_2 & M_{\bar{y}2} & M_{\bar{z}2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ N_n & V_{\bar{y}n} & V_{\bar{z}n} & T_n & M_{\bar{y}n} & M_{\bar{z}n} \end{bmatrix}$$

The section forces is based on the basic Euler-Bernoulli beam equations, Equation 3.64.

$$EA\frac{d^2\bar{u}}{d\bar{x}^2} + q_{\bar{x}} = 0 \tag{3.64}$$

$$EI_z\frac{d^4\bar{v}}{d\bar{x}^4} - q_{\bar{y}} = 0$$

$$EI_y\frac{d^4\bar{w}}{d\bar{x}^4} - q_{\bar{z}} = 0$$

$$GK_v\frac{d^4\bar{\varphi}}{d\bar{x}^4} + q_{\bar{w}} = 0$$

The displacements along the beam element are obtained as the sum of the homogeneous and the particular solutions, Equation 3.65.

$$\boldsymbol{u} = \begin{bmatrix} \bar{u}(\bar{x}) \\ \bar{v}(\bar{x}) \\ \bar{w}(\bar{x}) \\ \bar{\varphi}(\bar{x}) \end{bmatrix} = \boldsymbol{u}_h + \boldsymbol{u}_p \tag{3.65}$$

Where the homogeneous solution is

$$\boldsymbol{u}_h = \bar{\boldsymbol{N}}\boldsymbol{C}^{-1}\boldsymbol{G}\boldsymbol{a}^e$$

and the particular solution

$$\boldsymbol{u}_p = \begin{bmatrix} \bar{u}_p(\bar{x}) \\ \bar{v}_p(\bar{x}) \\ \bar{w}_p(\bar{x}) \\ \bar{\varphi}_p(\bar{x}) \end{bmatrix} = \begin{bmatrix} \frac{q_{\bar{x}} L \bar{x}}{2EA}\left(1 - \frac{\bar{x}}{L}\right) \\ \frac{q_{\bar{y}} L^2 \bar{x}^2}{24EI_z}\left(1 - \frac{\bar{x}}{L}\right)^2 \\ \frac{q_{\bar{z}} L^2 \bar{x}^2}{24EI_y}\left(1 - \frac{\bar{x}}{L}\right)^2 \\ \frac{q_{\bar{w}} L \bar{x}}{2GK_v}\left(1 - \frac{\bar{x}}{L}\right) \end{bmatrix}$$

and

$$\bar{\boldsymbol{N}} = \begin{bmatrix} 1 & \bar{x} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \bar{x} & \bar{x}^2 & \bar{x}^3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \bar{x} \end{bmatrix}$$

$$\boldsymbol{C} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & L & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & L & L^2 & L^3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & L & L^2 & L^3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & L \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2L & 3L^2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2L & 3L^2 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\boldsymbol{a}^e = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_1 2 \end{bmatrix}$$

Finally the section forces are obtained from Equation 3.66.

$$N = EA\frac{d\bar{u}}{d\bar{x}} \tag{3.66}$$

$$V_{\bar{y}} = -EI_z\frac{d^3\bar{v}}{d\bar{x}^3}$$

$$V_{\bar{z}} = -EI_y\frac{d^3\bar{w}}{d\bar{x}^3}$$

$$T = GK_v\frac{d\bar{\varphi}}{d\bar{x}}$$

$$M_{\bar{y}} = -EI_y\frac{d^2\bar{w}}{d\bar{x}^2}$$

$$M_{\bar{z}} = -EI_z\frac{d^2\bar{v}}{d\bar{x}^2}$$

4
Methodology

4.1 Geometries and structures

Based on the literature survey, three main structural concepts were selected and evaluated. Instead of analyzing a structure with the dimension of a full test specimen, each structure was designed as a column of unit cells where the unit cell was based on one geometrical concept.
A unit cell is the smallest structure which can be placed repetitively in any direction to form the lattice structure or analysis model. Analyzing a column of unit cells compared to a full model is preferable since it significantly reduces computational time. For computationally heavy FEM analyses, it means more tests and more numerical tests can be performed for each structure. A column of unit cells is used to model the entire material by symmetry conditions.
However, some general assumptions must be made in order for this setup to be applicable. It is assumed that one column of unit cell is small compared to the number of columns required to build the full model of the considered material volume. A Dirichlet boundary condition is applied to the outermost facing nodes, simulating neighbouring cells which prevents any movement in $xy-$plane and only allow movement in z-direction.

4.1.1 Cellulose

The cellulose, which is described in subsection 2.3.3, is the backbone of the cell wall and contributes with stability, stiffness and strength to the wood. The polymer is strictly hierarchical, Figure 4.1 shows a micro graph of wood and illustrates each cell walls surrounding each cell and how they overlap each other.

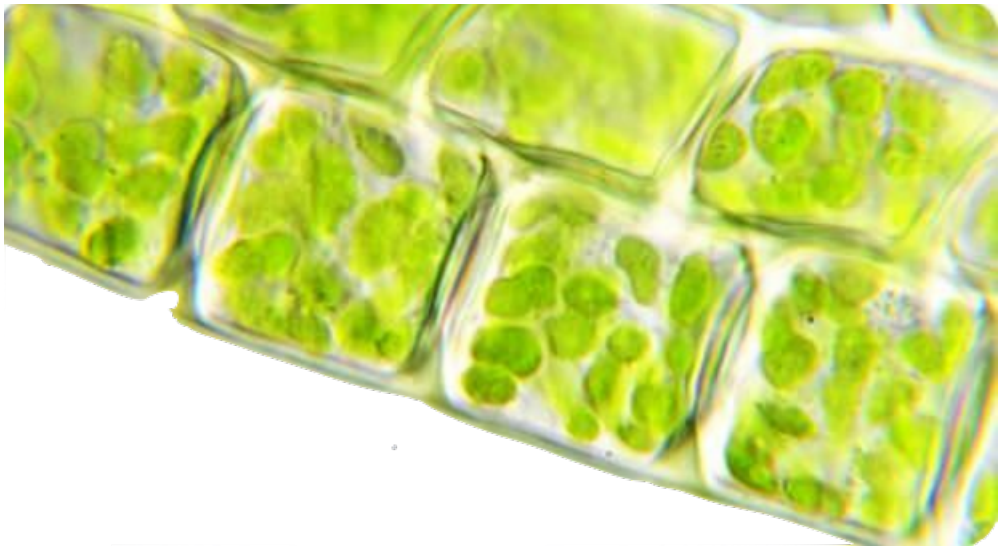


Figure 4.1: Image of plant cells taken with a light microscope where one can clearly see the green chloroplast and the cell wall around each cell [25].

Cellulose in the cell wall of plants inspired the design of the unit cell structure *cellulose*. The *cellulose* unit cell consists of two cubes stacked over each other with an offset. The offset is only in one direction to gain some strength, compared to if the cubes were placed with an offset in both horizontal ways. The offset is designed in order to transfer loads between the beams and plastically deform them in a somewhat smooth fashion. Because of the rigid joints between the beams the buckling mode will transfer from one layer to the other. This structure does not have any diagonal beams, hence making the unit cell quite hollow compared to the other structures. However it has vertical beams and the structure will be stiffer than diagonally placed beams up to a critical load. The two cubes are seen as one unit cell since in pair, it can be repeated in any direction. A plot of the cellulose unit cell is displayed in Figure 4.2.

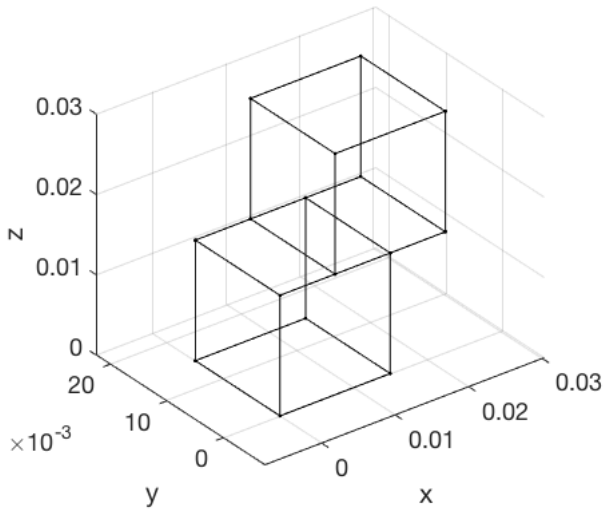


Figure 4.2: Cellulose unit cell.

4.1.2 Tetrahedron

In subsection 2.3.1, the composition of human bone is described. In every day life, our hips, spine, and pelvis are subjected to mechanical stresses due to physical activities. The strength that keeps the bones from breaking is provided by the trabeculae. A geometric figure that approximately resembles the stochastic pattern in trabeculae is the tetrahedron. Tetrahedron is a polyhedron (three-dimensional solid consisting of polygons, often joined at their edges) with four faces. Depending on how its dimensions are varying, the tetrahedron is either a regular tetrahedron or a non-regular tetrahedron. A regular tetrahedron has all faces congruent to an equilateral triangle, i.e. all sides are of the same dimensions whereas a non-regular has not [26]. With the trabeculae in mind and tetrahedron as a base, the tetrahedron unit cell was constructed. It consists of eight tetrahedrons and has 90 degrees rotating base plates in the middle layer in order to make the structure repeatable in all directions, see Figure 4.3. The main characteristics of the Tetrahedron structure is many but short beams. Each connection node has a multitude of beams connecting to it. The structure raises zigzagging in y-z-direction and has no vertical beams

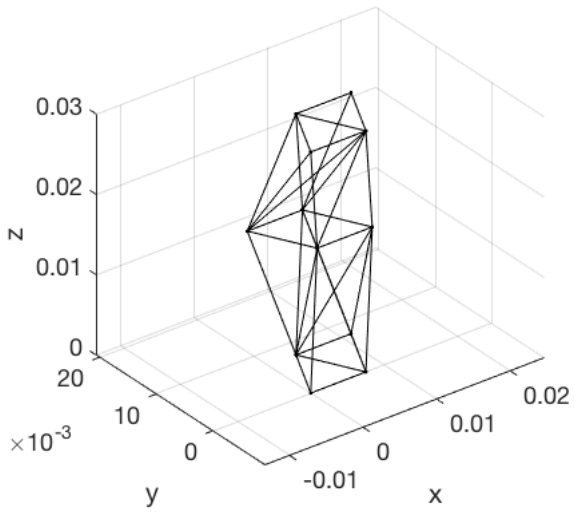


Figure 4.3: Tetrahedron unit cell.

4.1.3 Pyramid

The geometric figure pyramid is a polyhedron, like the tetrahedron it also has some similarity to the stochastic pattern of the trabuculae. Unlike the tetrahedron it is easier to construct a unit cell from the pyramid since it can have a base of four corners. A pyramid has its base as a polygon and the other faces as triangles meeting at a common vertex, known as the apex. The pyramid can be of various shapes since the polygon can be of n-sides, but the regular pyramid has its base as a regular polygon ($n \in [3, 4, 5]$) and is also a right pyramid, which means that the apex lies right above the centroid of the base [27]. A plot of the unit cell is displayed in Figure 4.4.

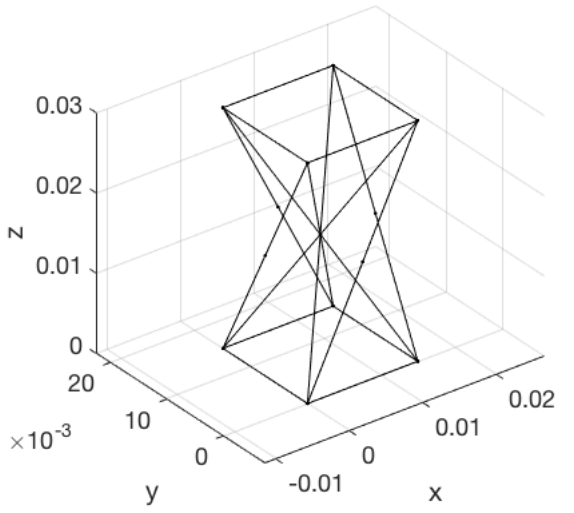


Figure 4.4: Pyramid unit cell.

4.2 Material

The material selection is limited to commercially available 3D printable polymers to allow future validation of the models against tests on printed objects. The following four polymers were selected for this study: ABS (acrylonitrile butadiene styrene), PLA (polylactide), Nylon and TPU (thermoplastic polyurethane). The material characteristics are presented in Figure 4.5 and relevant properties are displayed in Table 4.1. In order to absorb energy the material should ideally be able to withstand both high stresses and strains, since the area under the stress strain curve is the energy. Both ABS and PLA are too brittle. Brittle materials will not result in a smooth deceleration nor be able to absorb any significant amount of energy. TPU is very soft and can withstand large strains, but with low stresses the deformation length needs to be very large in order to compensate for this. Nylon cannot handle as large stresses as ABS or PLA but behaves similar to an elastic perfectly plastic material, which is good for energy absorption.

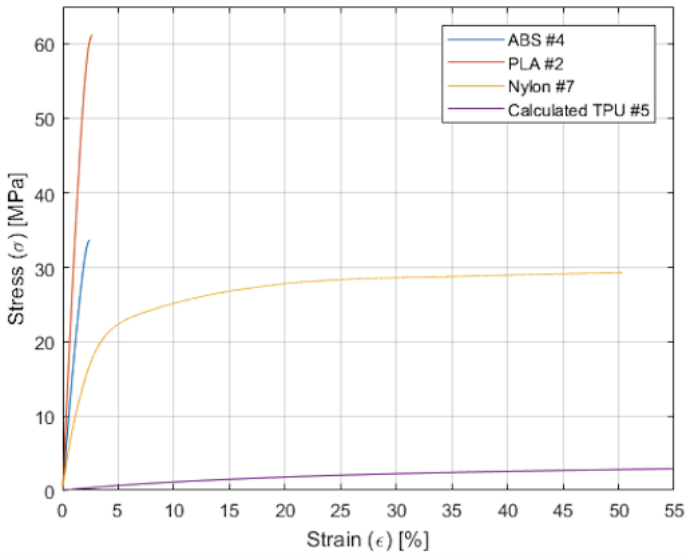


Figure 4.5: Material comparison for 3D printable polymers [28].

Therefore, nylon is selected as material for all geometries and its material data is implemented in the code. The behaviour of Nylon is simplified as being an elastic perfectly plastic material.

Table 4.1: Material properties for 3D printable polymers [28].

Property	ABS	PLA	Nylon	TPU
Density [kg/m^3]	1060	1210	1160	1120
Young's modulus [MPa]	1031	3310	940	12
Shear modulus [MPa]	318.9	2400	359.7	8.6
Yield strength [MPa]	42.5	110	31	4

4.3 Applied force

The bicycle helmet drop test needs to be converted into an averaged force in order to simulate the compression test. This is done using the conservation of energy principle for a falling object combined with the work-energy principle. European bicycle helmets need to withstand a drop a weight of 5 kg from 1.5 m [4]. The mass and height is denoted m and t, respectively. The energy absorbent structure, that is to be designed, has a defined thickness t that must absorb the impact. The test setup is illustrated in Figure 4.6.

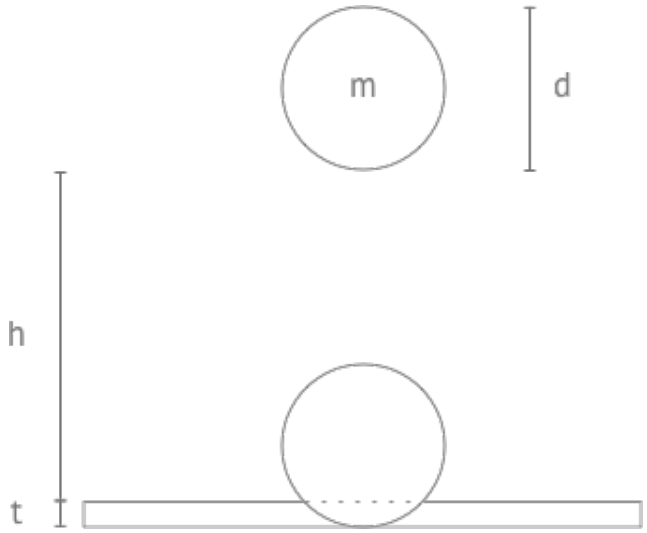


Figure 4.6: Drop test setup.

The kinetic energy, E_k, just before the impact is equal to its gravitational potential energy, E_p, at the drop height Equation 4.1.

$$E_k = \frac{1}{2}mv^2 = E_p = mg(h+t) \tag{4.1}$$

The impact force is calculated using the work-energy principle, Equation 4.2. The change in the kinetic energy of the object is equal to the work done on the object. Since the impact on the helmet comes to a stand still, the final kinetic energy, $E_{k,final} = 0$.

$$W_{net} = E_{k,final} - E_{k,initial} = -E_k \tag{4.2}$$

$$W_{net} = Ft \Rightarrow F = -mg(1 + h/t) \tag{4.3}$$

In order to calculate the applied load onto one unit cell, the contact area needs to be determined. A bicycle helmet has a mean radius of 160 mm [29], denoted r. The helmet is assumed to be deformed its entire thickness, this gives the contact area A as Equation 4.4.

$$A = \pi r^2 cos^2(sin^{-1}\Big(1 - \frac{t}{r}\Big)) \tag{4.4}$$

This results in an applied stress, Equation 4.5.

$$\sigma_{applied} = \frac{F}{A} = \frac{-mg(1 + h/t)}{\pi r^2 cos^2(sin^{-1}(1 - t/r))} \tag{4.5}$$

Finally, the applied force on one unit cell is the applied stress multiplied with the unit cell's top area. For all three geometries, the top area is the width in square. Figure 4.7 depicts the relation between applied force on one unit cell and the width of the unit cell.

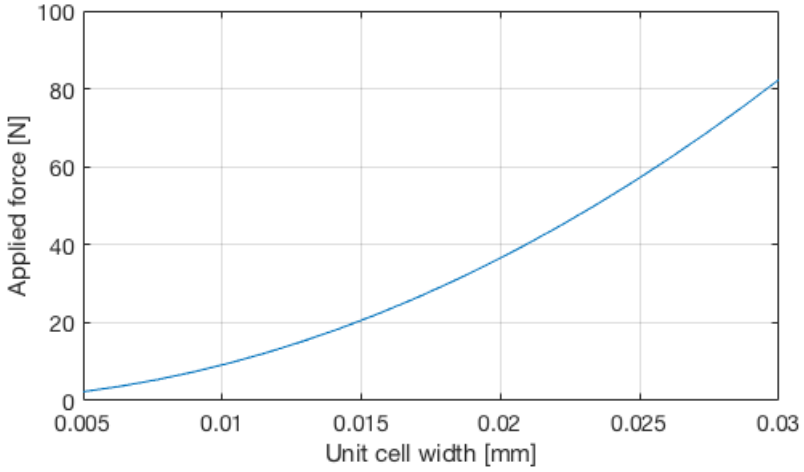


Figure 4.7: Applied force depending on unit cell width.

4.4 Design variables

In order to describe and compare energy absorption for different geometries for each concept, four design variables are introduced: Number of unit cells, width of the unit cell, beam radius and a height factor. Each design variable is defined with an individual range defining its minimum and maximum value and resolution.

Number of unit cells

Each geometry is built up by unit cells. Unit cells are the basic building blocks of the structure which defines its core characteristics. the structure that is the core of its characteristics, see Figures 4.2 - 4.4. This thesis evaluates three different unit cells named after their conceptual look; tetrahedron, pyramid and cellular. The geometries varies from one to ten unit cells. More unit cells equals more beams and higher degrees of freedom in the problem resulting in longer computational time. The higher limit of 10 unit cells for a geometry is set due to manufacturing limitations. More unit cells will generate a denser structure, which is difficult to print.

Unit cell width

The width of a unit cell defines its footprint on the base plate. By varying the width, the degrees of freedom remain the same and the calculation time likewise. Increased width leads to higher mass and a higher applied force. The width has a range from 5 mm to 30 mm, with a resolution matching that of the 3D printer. The design range is determined from multiple tests from the script, displaying high specific energy absorption within the range.

Beam radius

Each beam is circular and the radius is a key design variable. The radius heavily influences the critical load for buckling in the beam and the mass of the structure. The minimum radius is 0.05 mm, since it is the finest the manufacturing can produce. The maximum radius is dynamically calculated with the combination of other design variables to make sure no beam is colliding with another. The resolution is the same as that of the 3D printer's.

Height factor

The height factor is a dimensionless integer defining the height of a unit cell relative to other unit cells. The range is between one and four. The lower limit generates a structure where all unit cells are of the same height. The upper limit generates each unit cell four times higher than the unit cell below it. Tests shows that larger height factor generates a too small first unit cell which is not printable. The resolution is experimentally set to 0.1. Finer resolution would increase computational time but the influence of the solution is minimal below this resolution.

4.5 Identification of design space

There are three main sections in the script used in this thesis; defining the design variable range, perform tests from a rough mesh of combinations to find a suitable start guess and lastly fine-tune the start guess using a optimization algorithm. The design variable range is manually defined for each design variable and geometry. Depending on the geometry, the range varies for what is possible to manufacture. For example, the tetrahedron has multiple diagonal beam elements and is more dense than cellulose. Hence the tetrahedron cannot have as high radius nor height factor as cellulose. The sections of the script are shown in Figure 4.8

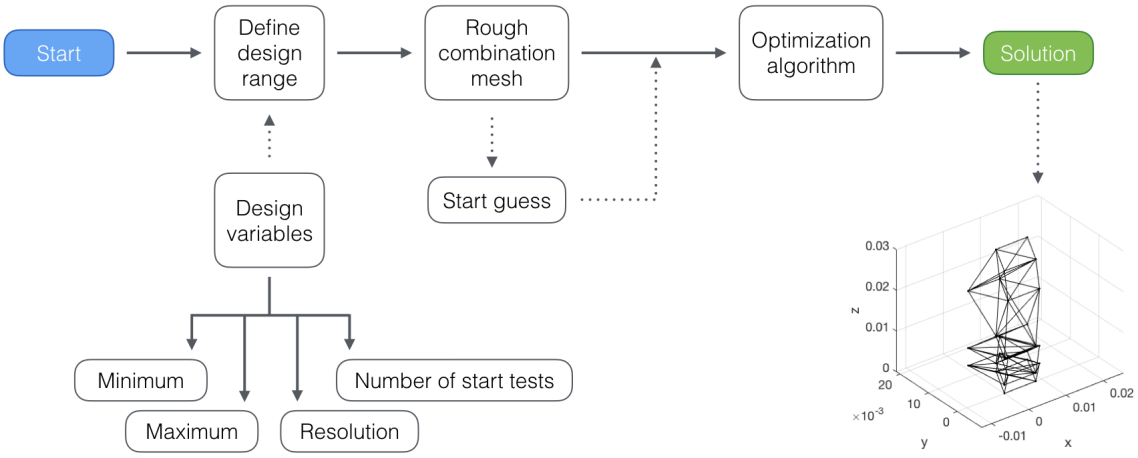


Figure 4.8: Overview schematics of the main steps in the code.

4.5.1 Rough combination mesh

The rough combination mesh allows for an overview of the design variables behaviour and how they accommodate with one another. The rough mesh is used to screen the entire design variable space and define what combinations generate high specific energy absorption and what combinations that does not. The rough mesh is illustrated in Figure 4.9 for a system of two design variables D_1 and D_2, each with a range of 0 to 1 and a resolution of 0.02. The result is a function $f(D_1, D_2)$ presented colorized on the z-axis. The combination of D_1 and D_2 that generates the highest result f is sought. The most straight forward approach would be to test all combinations and evaluate them against each other, as the graph displays. However, this would result a number of 2,500 unique combinations.

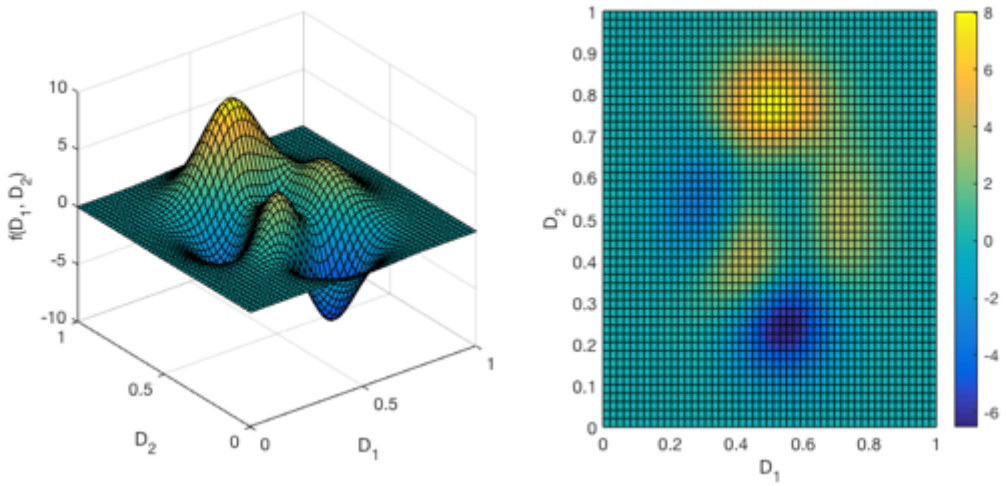


Figure 4.9: Two design variables illustrating the result of all combinations.

An alternative to calculating all unique combinations, presented in Section 4.5.2, is to compare the results from a rough mesh of the system, see Figure 4.10. If the resolution of the rough mesh is coarse, more tests are needed in order to evaluate all

combinations within that design space. The peak value is marked in red, highlighting the maximum value of f and is the start guess used in the optimization algorithm.

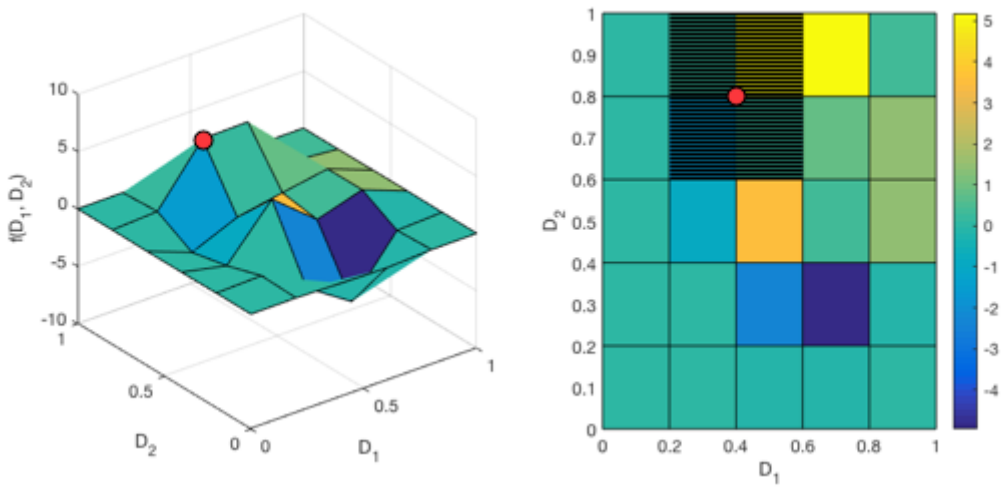


Figure 4.10: Coarse mesh of two design variables resulting in a large design range for the fine tuning.

A finer mesh in the initial rough combination test, Figure 4.11, would take longer to perform but needs fewer tests to fill the design space surrounding the peak value. With a finer grid, the probability of missing a local maximum decreases.

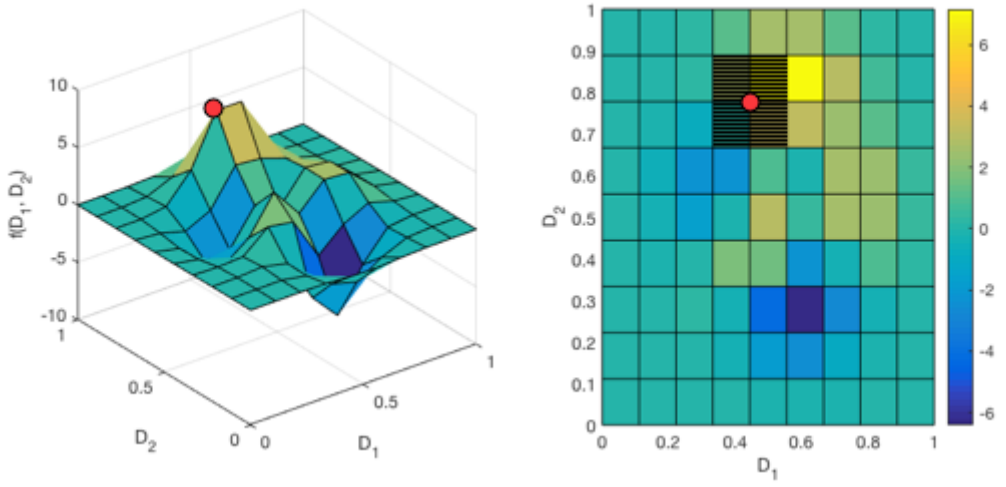


Figure 4.11: Fine mesh of two design variables resulting in a smaller design range for the fine tuning.

The design variables mesh size is depending on its influence on the evaluation function. For example, the beam radius affects the mass squared and the critical buckling load quadratic. Hence a finer mesh size should be used for the beam radius. The width on the other hand does not influence the specific energy absorption as drastically and a more coarse mesh can be applied.

4.5.2 Optimization algorithm

In this thesis, three geometries are studied to find a combination of four design variables in order to find the combination which has the highest specific energy absorption. The design variables are number of unit cells, unit cell width, beam radius and height factor. The material parameter and combined structure height are predefined. All design variables have a identified design range. By using a traditional method, all combinations within the variables design range need to be tested and compared in order to identify the values that give the highest specific energy absorption. However, this is a very time consuming procedure. If the four design parameters are tested with 100 evenly spaced steps this would result in 100,000,000 different tests. On a 1.4 GHz Intel Core i5 one test takes on average 5 seconds, this would result in 15 years of computational time to find the best combination. A more practical approach for saving computational time is therefore needed.
We start by assuming that the sought combinations are not too sensitive to change in any one of the parameters, meaning that a good combination is still quite good even if some parameters are slightly changed. With this in mind, we do not need to test a fine mesh for all design parameters in order to find a pattern and locate some combinations that tend to be more energy absorbent than others. When locating a combination from this rough resolution it is used as a start guess. Each design variable can then be tested separately, see Figure 4.12. If any of the design variables differs from the start guess, the procedure starts over again with the updated start guess. Note that the design variable only needs to be tested within the design range not covered in the initial rough resolution combination test, as illustrated in Section 4.5.1.

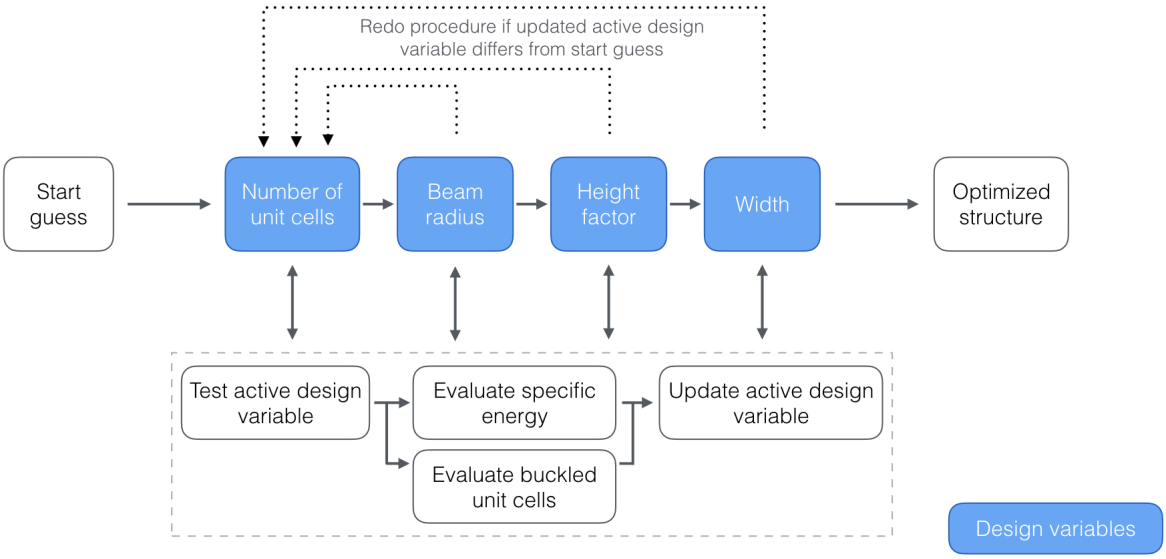


Figure 4.12: Schematics of the code structure for the optimization algorithm.

As illustrated in Figure 4.12, the same procedure of evaluating the design parameters apply to all. When an active design parameter is tested, it steps in a local range from some minimum value to a maximum value. Whichever yields the highest specific energy absorption is kept as the new value. It is also important to evaluate

the number of buckled levels since this directly correlate to the deceleration of an impact. Hence if two similar values are found, the one with most buckled levels will be selected. In order to evaluate how much more efficient this method is, we assume that we have X number of design variables. Each design variable has an individual number of tests associated with it. The number of tests per design variable is collected in a vector $\boldsymbol{N}$.

$$\boldsymbol{N} = [N_1, N_2, ..., N_{X-1}, N_X] \tag{4.6}$$

One way to find the best combination is to test all combinations. This would result in a total number of combinations C.

$$C = N_1 \cdot N_2 \cdot ... \cdot N_{X-1} \cdot N_X \tag{4.7}$$

Now, assume that a rough resolution is applied to each design variable. The resolutions are collected in a vector $\boldsymbol{R}$.

$$\boldsymbol{R} = [R_1, R_2, ..., R_{X-1}, R_X] \tag{4.8}$$

For all individual design variables, the rough resolution is lower than the total number of tests connected to it.

$$R_i < N_i, \quad \forall \quad 1 \leq i \leq X \tag{4.9}$$

With this, the number of combinations to find the start guess, C_R, is:

$$C_R = R_1 \cdot R_2 \cdot ... \cdot R_{X-1} \cdot R_X << C \tag{4.10}$$

The rest combinations between the rough mesh is the quote between all combinations and the ones performed, individual for all design variables. This rest resolution is gathered in the vector $\boldsymbol{R}_R$.

$$\boldsymbol{R}_R = \frac{\boldsymbol{N}}{\boldsymbol{R}} = \left[\frac{N_1}{R_1}, \frac{N_2}{R_2}, ..., \frac{N_{X-1}}{R_{X-1}}, \frac{N_X}{R_X}\right] \tag{4.11}$$

Each element in the rest resolution is less than the number of tests for that design variable.

$$R_{R,i} < N_i \quad \forall \quad 1 \leq i \leq X \tag{4.12}$$

In the optimization algorithm, each design variable is evaluated one at a time with the other variables kept constant. If a combination with higher specific energy absorption is found for the evaluated design variable, the start guess is updated with the new value and the script starts over again. This means that with a good start guess each design variable can be evaluated without finding a better combination, hence resulting in few tests being performed. On the other hand, the start guess can be updated multiple times for every design variable and restart the test, resulting in more tests. In the best case scenario, the start guess is the best combination.

$$T_{best} = \sum_{i=1}^{X} R_{R,i} \tag{4.13}$$

In the worst case scenario, every design variable resets the script with an updated start guess.

$$T_{worst} = \sum_{i=1}^{x} \sum_{j=1}^{i} R_{R,i} \tag{4.14}$$

If all design variables has the same number of tests and the same rough mesh resolution, the worst case scenario would be:

$$T_{worst} = \sum_{i=1}^{X} R_R \cdot i = \frac{R_R X}{2}(X+1), \quad \begin{cases} N_1 = N_2 = ... = N_{X-1} = N_X \\ R_1 = R_2 = ... = R_{X-1} = R_X \end{cases} \tag{4.15}$$

The number of tests performed, T, will be between the best and worst case scenario.

$$C_R + T_{best} \leq T \leq C_R + T_{worst} \tag{4.16}$$

In order to give a direction of efficiency for the optimization algorithm, all design variables are assumed to have the same resolution and number of tests. There are 4 design variables, each with 1,000 tests. This results in 1,000,000,000,000 different combinations. In Table 4.2 the rough resolution, R, is tested from 1 to 10 in order to calculate the number of tests to be perform.

Table 4.2: Different rough resolutions and it's effects on best and worst case scenario for number of tests.

R	C_R	R_R	T_{best}	T_{worst}
1	1	1000	4001	10001
2	16	500	2016	5016
3	81	334	1417	3421
4	256	250	1256	2756
5	625	200	1425	2625
6	1296	167	1964	2966
7	2401	143	2973	3831
8	4096	125	4596	5346
9	6561	112	7009	7681
10	10000	100	10400	11000

Figure 4.13 displays the best and worst case scenario from Equation 4.13 and 4.14. With this particular setup, the program would need to run the fewest tests if a rough resolution of 5 is used.

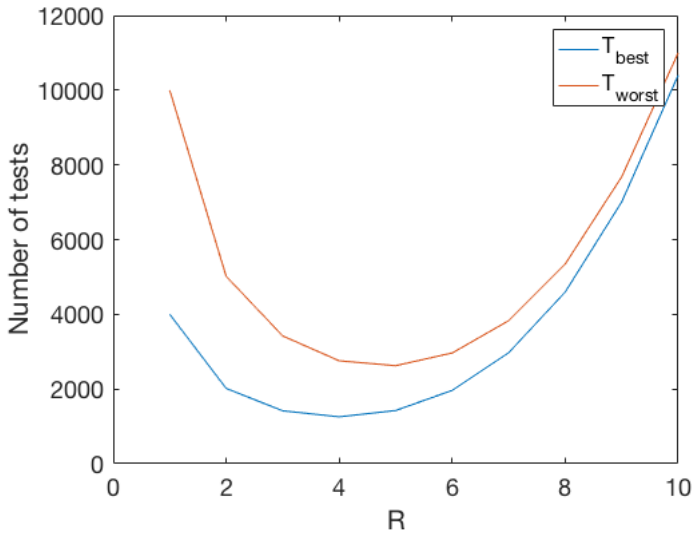


Figure 4.13: Best and worst case scenario for number of tests performed, assumed the four design variables have the same design range and resolution.

4.6 Compression test simulation

The compression test simulation is a system of scripts which builds the model, applies a force, solves the FEM problem, evaluates buckling and calculates the energy absorption with fixed values for the design variables. Schematics of the system is presented in Figure 4.14. The following subsections will describe in detail each major part of the compression test simulation. See Appendix B for the Matlab code.

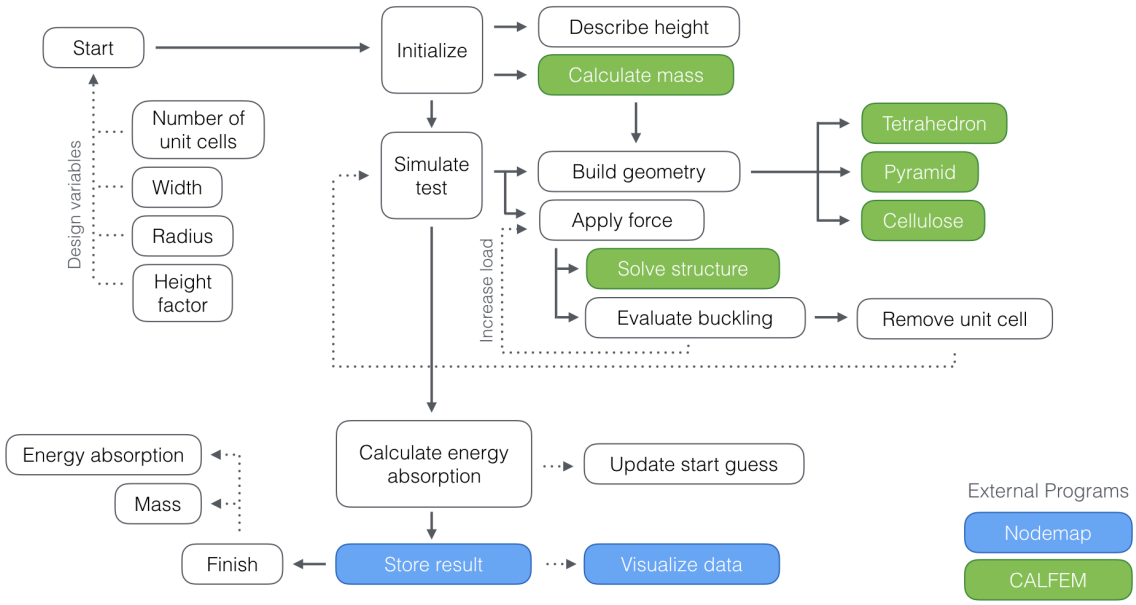


Figure 4.14: Schematics of the test simulation script.

4.6.1 Determination of unit cell height

In order to build the structure, the height of each unit cell needs to be determined. The unit cell height is dependent on the total height h_{tot}, number of unit cells N and the height factor h_f. Since each unit cell's height is described as the height of the unit cell below multiplied with the height factor, the first unit cell height needs to be determined. With the first unit cell height, h_1, the rest are defined as:

$$h_2 = h_1 h_f^1$$

$$h_3 = h_1 h_f^2$$

$$h_{N-1} = h_1 h_f^{N-2}$$

$$h_N = h_1 h_f^{N-1}$$

$$\Rightarrow h_i = h_1 h_f^{i-1} \quad \forall \quad 1 \leq i \leq N \tag{4.17}$$

The sum of all unit cell heights is the total height of the structure. With this, the first unit cell height is solved as:

$$\sum_{i=1}^{N} h_i = \sum_{i=1}^{N} h_1 h_f^{i-1} = h_1 \sum_{i=1}^{N} h_f^{i-1} = h_{tot} \tag{4.18}$$

$$h_1 = \frac{h_{tot}}{\sum_{i=1}^{N} h_f^{i-1}} \tag{4.19}$$

Finally, each unit cell height is calculated as:

$$h_i = \frac{h_{tot} h_f^{i-1}}{\sum_{j=1}^{N} h_f^{j-1}} \quad \forall \quad 1 \leq i \leq N \tag{4.20}$$

In Figure 5.10, the affects of height factor 1, 2 and 3 are illustrated for each geometry with number of unit cells fixed to 3 and the total height is 30 mm.

4.6.2 Build geometry

There is only one function, *Build geometry*, that calls the individual geometry constructors. This allows for further expansion and addition of other geometries to be implemented into the code without adjusting large proportions of the code base. The build geometry functions constructs a system of nodes depending on the chosen geometry, width and unit cell height. The function draws elements between specific nodes, making the model take its desired shape. An illustration of the build procedure is displayed in Figure 4.16. This procedure returns coordinates for all nodes, the element degrees of freedom matrix, degrees of freedom for each node, force vector and boundary conditions. All output variables are passed into the finite element problem, Section 3.2.

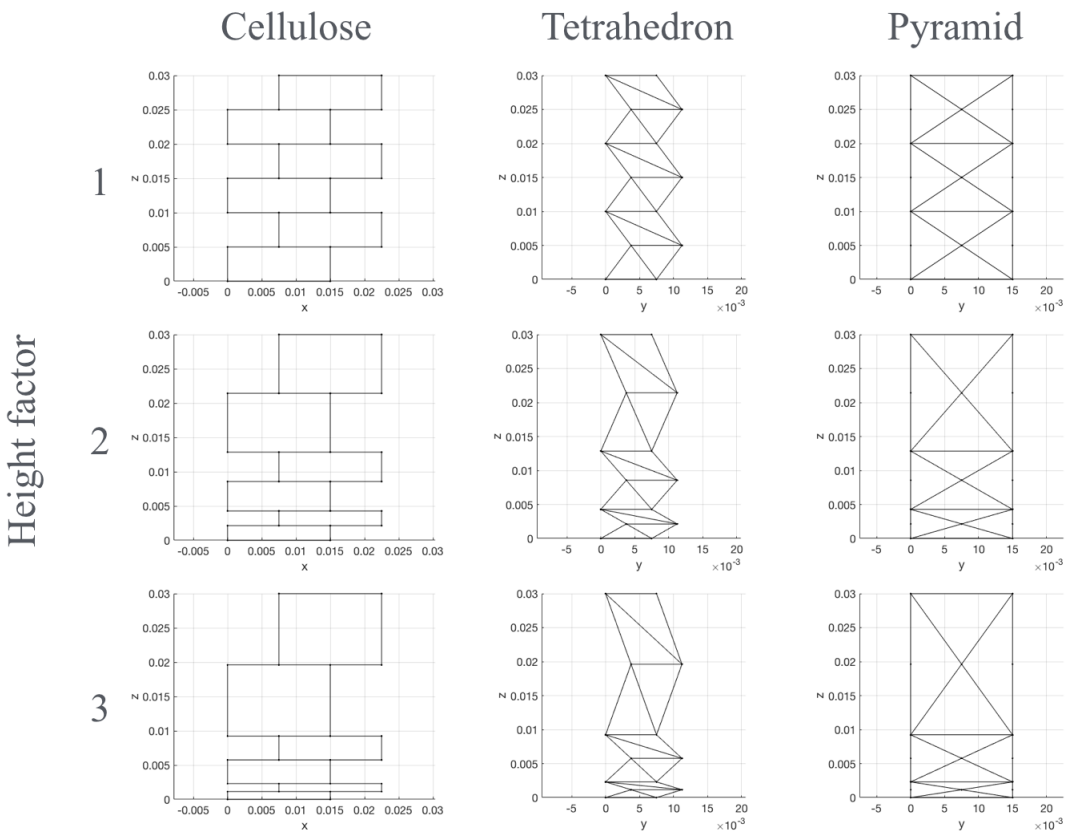


Figure 4.15: Effects of height factor.

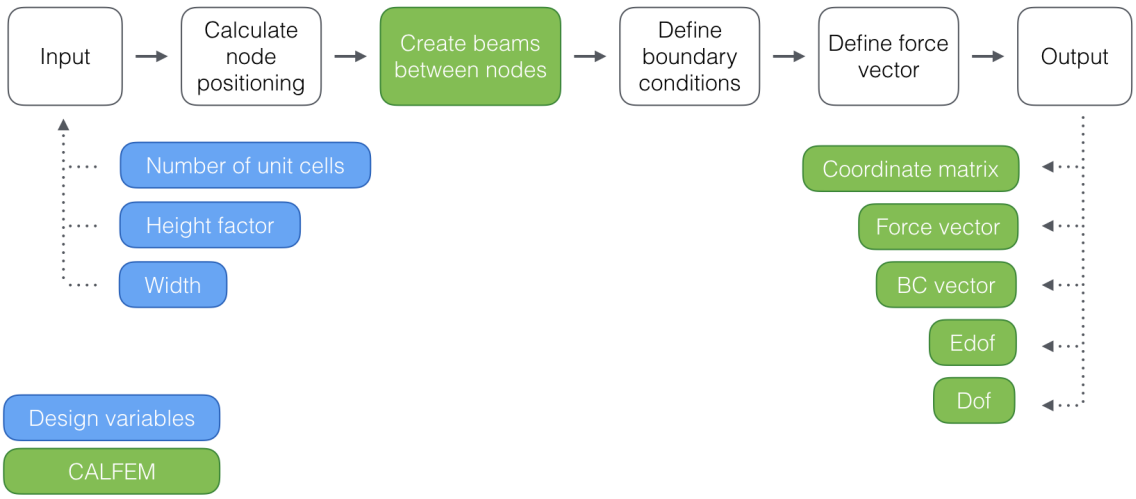


Figure 4.16: Schematics over the procedure of building a geometry.

4.6.3 Buckling

All beam deformations and end forces are evaluated and exported from the FEM solution into the buckling analysis. When buckling is studied, each beam is evaluated individually within the structure. The undeformed beam length, l_0, is calculated

with Equation 4.21.

$$l_0 = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{4.21}$$

Similarly, the deformed beam length, l, is calculated with the beam ends displacements taken into account. From the deformed and undeformed beam length, the stress, σ, is calculated using Hooke's Law, Equation 4.22. The material is assumed to be elastic perfectly plastic, see Figure 4.5.

$$\sigma = \begin{cases} E\epsilon & \text{if} \quad E\epsilon < \sigma_y \\ \sigma_y & \text{else} \end{cases} \tag{4.22}$$

The strain, ϵ is calculated as

$$\epsilon = \frac{l_0 - l}{l} \tag{4.23}$$

The axial force, F, acting on the beam is calculated from the stress and the cross sectional area of the cylindrical beam.

$$F = \pi\sigma r^2 \tag{4.24}$$

Lastly, each beam is evaluated against buckling. In order to determine which buckling case to use, a physical compression test is performed, see Figure 4.17. When zooming into one beam of the lower unit cell, the buckling case is displayed clearly; clamped in both ends, due to the vertical angle of which the beam connects onto the horizontal beams. The critical load, P_k, is calculates from Equation 4.25.

$$P_k = \frac{4\pi^2 EI}{l^2} \tag{4.25}$$

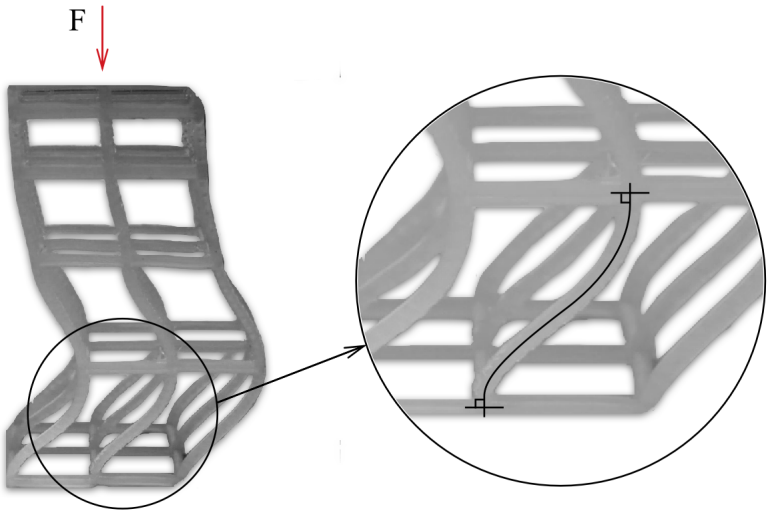


Figure 4.17: Physical compression test for evaluating buckling case of a nylon structure.

If the axial force in any beam exceeds the critical buckling load, the buckled unit cell is removed since its energy absorption properties has decreased significantly once buckled, see Section 2.6.

4.6.4 Specific energy absorption

The energy is calculated from the integral of the force-displacement curve for a simulated compression test. In Figure 4.18, the force is plotted against the deflection for a 5 unit cell tetrahedron structure. Each peak marks the maximum force that a particular unit cell is able to withstand before buckling. Since the height factor is always greater than 1 for the structures, the topmost unit cell will buckle first, followed by the second topmost etc.

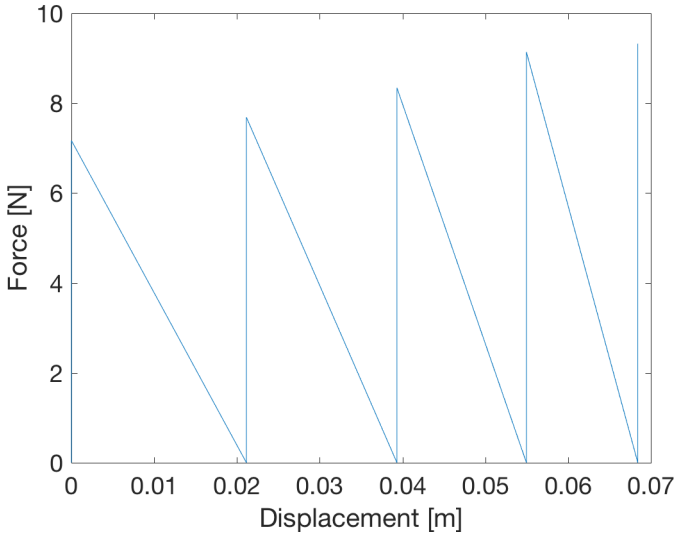


Figure 4.18: Force and displacement for a compression simulation test of a tetrahedral with 5 unit cells.

Since the script calculates displacement with a certain force step, the integral needs to be approximately calculated as the sum for all step sizes. Assume n number of points in the graph for the force F and the displacement δ, the energy E is calculated from Equation 4.26

$$E = \int_0^{\delta_{max}} F d\delta = \sum_{i=1}^{n-1} \frac{F_i + F_{i+1}}{2}(\delta_{i+1} - \delta_i) \tag{4.26}$$

In order to evaluate and compare the energy absorbing efficiency, the energy is normalized with the mass of the geometry. The mass, m_{tot}, is calculated as the sum of all beam masses, m, constructing the geometry. Assume N_{beams} number of beams in the structure, with radius r and density ρ.

$$m_{tot} = \sum_{i=1}^{N_{beams}} m_i \tag{4.27}$$

Each beam is circular and has an individual length, l_i.

$$m_i = \pi r^2 l_i \rho \tag{4.28}$$

The length of a beam is calculated from its start and end coordinates.

$$l_i = \sqrt{(x_i^{(1)} - x_i^{(2)})^2 + (y_i^{(1)} - y_i^{(2)})^2 + (z_i^{(1)} - z_i^{(2)})^2} \tag{4.29}$$

Lastly the specific energy absorption is calculated with Equation 4.30.

$$E_s = \frac{E}{m_{tot}} = \frac{\sum_{j=1}^{n-1} \frac{F_j + F_{j+1}}{2} (\delta_{j+1} - \delta_j)}{\sum_{i=1}^{N_{beams}} \pi r^2 \rho \sqrt{(x_i^{(1)} - x_i^{(2)})^2 + (y_i^{(1)} - y_i^{(2)})^2 + (z_i^{(1)} - z_i^{(2)})^2}} \tag{4.30}$$

If the active test results in a higher specific energy absorption than previous tests, the combination is stored as the updated start guess.

5
Results and discussion

5.1 Design variable range and test setup

Since the number of tests are highly influenced on the design range and resolution for each design variable, it is important to have a good setup. If the design range is too large and a rough mesh is used, the start guess for the optimization algorithm may not be good enough. After multiple initial tests and mapping of design variables behaviour, the setup range and resolution for each parameter is determined and presented in Table 5.1.

Table 5.1: Design variable range for test setup.

		Tetrahedron	Cellulose	Pyramid
Number of unit cells	Minimum	3	3	3
	Maximum	5	5	5
	Resolution	1	1	1
	Start tests	3	3	3
Beam radius [mm]	Minimum	0.5	0.5	0.5
	Maximum	2.0	3.0	2.0
	Resolution	0.01	0.01	0.01
	Start tests	10	10	10
Width [mm]	Minimum	10.0	10.0	10.0
	Maximum	30.0	30.0	30.0
	Resolution	0.1	1.0	1.0
	Start tests	3	3	3
Height factors	Minimum	1.1	1.1	1.1
	Maximum	3.0	3.0	3.0
	Resolution	0.01	0.05	0.05
	Start tests	5	5	5

5.2 Design variables impact on the mass

The minimum and maximum mass for each design variable is presented in Table 5.2. In order to demonstrate the influence on the mass from each design variable, the starting point is a fixed value for each variable: number of unit cells 3, width 20 mm, beam radius 2 mm and a height factor of 2. These constants are close to the found solutions for all three geometries and will serve as a reference guide where as

each design variable is evaluated separately. The design range for each variable has different influence on the total mass, which is a key characteristic for specific energy absorption since it is used to normalize the energy absorption.

Table 5.2: Design variables' influence on total mass.

	Mass [mg]	Tetrahedron	Cellulose	Pyramid
Number of unit cells	Minimum	11.3	7.8	10.4
	Maximum	28.2	28.9	28.4
Beam radius	Minimum	0.010	0.009	0.010
	Maximum	63.7	59.3	61.1
Width	Minimum	10.4	6.7	9.3
	Maximum	20.2	20.2	20.1
Height factors	Minimum	15.6	14.8	14.9
	Maximum	16.5	14.8	16.0

The beam radius has greatest impact on the the total mass. Due to it's influence on the beam's cross sectional area as a function of square it also effects the mass as a square function. This is clearly visible in Figure 5.1.

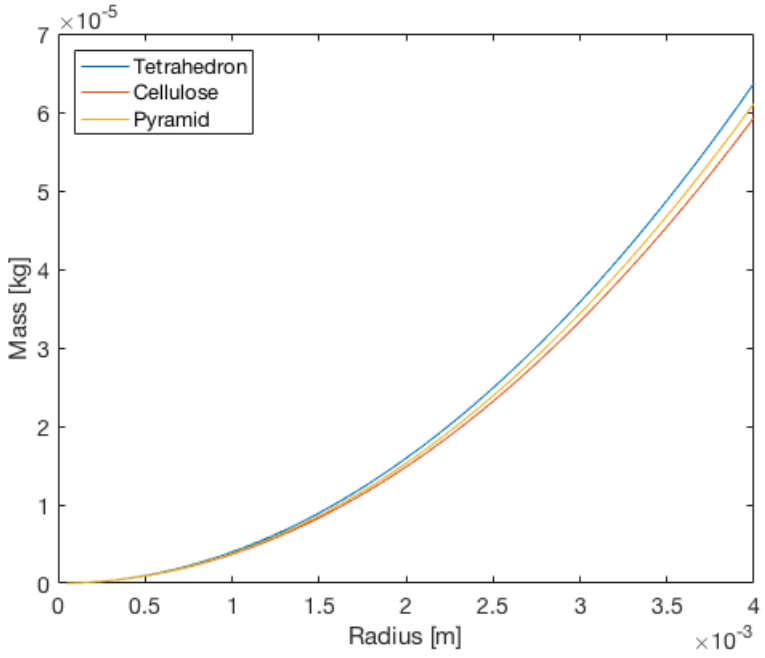


Figure 5.1: Mass dependency of beam radius.

The height factor impacts the mass the least with approximately 1 milligram, and for the cellulose structure it does not affect the weight at all. In the cellulose geometry the beams are either horizontally or vertically placed. By changing the height factor, the number of beams remains the same. The horizontal beams remains unchanged and with increasing height factor, the length the beams at the bottom becomes shorter, the beams at the top becomes longer. Hence, the mass remains the same for

the cellulose structure, see Figure 5.2. For the tetrahedron and pyramid structures, the mass as function of height factor shares similar shape. The amplitude of the tetrahedron is however higher than that of the pyramid. This is due to a greater number of beams in the tetrahedron, hence higher total mass to start with (assuming the same design variables). Both structures have tilted beams, hence changing the mass in an nonlinear manner.

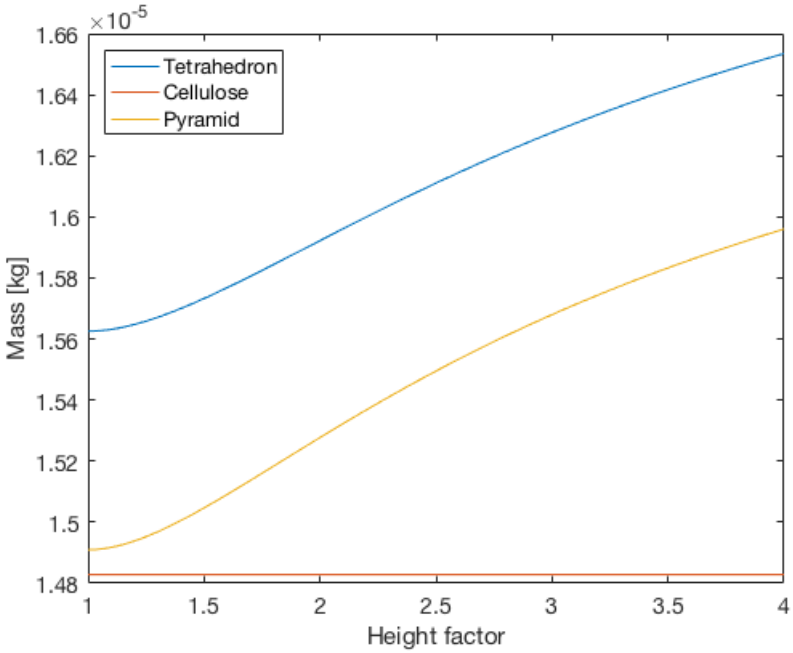


Figure 5.2: Mass dependency of height factor.

Both number of unit cells and width of the unit cells have similar mediocre impact on the mass in a linear manner, except for the cellulose that is linear. Small changes in either of these design variables will not drastically affect mass, see Figure 5.3.

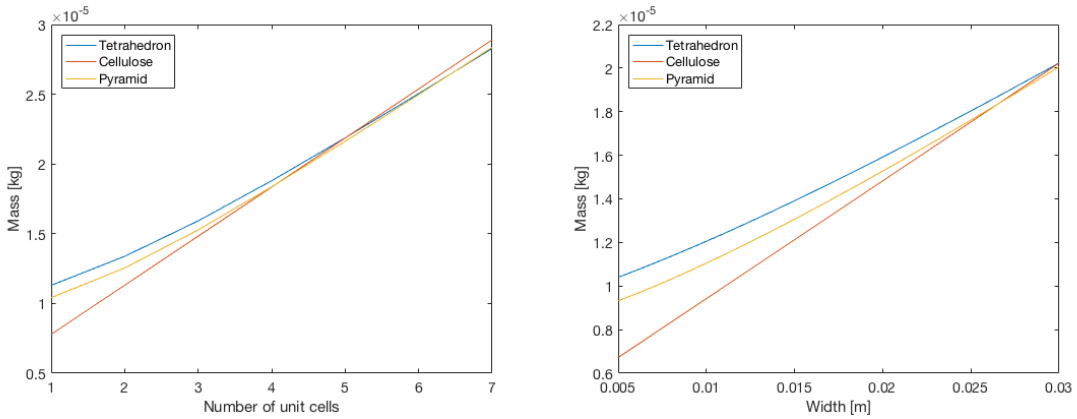


Figure 5.3: Mass dependency of number of unit cells (left image) and width (right image).

5.3 Screening results from rough mesh

The number of tests performed in the initialization stage is the factor of all start tests for each design variable, Table 5.1. Each geometry is evaluated separately and the combination with highest specific energy absorption is used in the optimization algorithm as start guess. In Figure 5.4, 5.5 and 5.6, the result from all combinations are presented as mass versus specific energy absorption. If any combination results in a too weak structure, i.e. not able to withstand the applied force, the combination is displayed as an asterisk (*) in the graph. All structures share some characteristics, clearly visible in the figures. With decreasing mass, the proportion of failed structures increases. The mass is highly influenced by the beam radius and number of unit cells. Deviation in radius affects the mass squared. Deviation in the number of unit cells results in the number of beams in the structure. Hence, low radius and few unit cells will result in a low mass. Few unit cells also generates longer beams, since the total height of the structure remains. The buckling evaluation is sensitive to both long and slender beams, resulting in buckling with low critical load and failed structures. Structures with high mass on the other hand, results in too stiff structures that will not absorb the energy but rather translate it to the ground boundary. High-mass structures generally have a large beam radius and a high number of beams, due to many unit cells. The opposite of the buckling case is reached, the beams are short and have a large radius resulting in a high critical load and few, or no, unit cells will buckle. There exist combinations that are too weak and absorbs large amounts of energy but fail to withstand the applied load, and there are combinations that are too strong, handling the applied force but do not manage to absorb significant amount of energy. In this solution domain, there is a possibility of a combination that are able to hold the applied force without buckling all unit cells, but at the same time absorb a large proportion of the energy by buckling the majority of the structures cells. The initialization procedure provides a guideline and start guess for such a combination.
In the model, the many combinations for which no unit cell buckle only absorb energy elastically, since plasticity is not evaluated. Elastic energy absorption is very low, compared to plastic, and results in a design range of 0.1 - 2 J/kg, making it seem like zero in the graphs. This thesis is constructed on the theory that buckling is the main contributor to energy absorption and evaluates this through the force versus displacement curve.

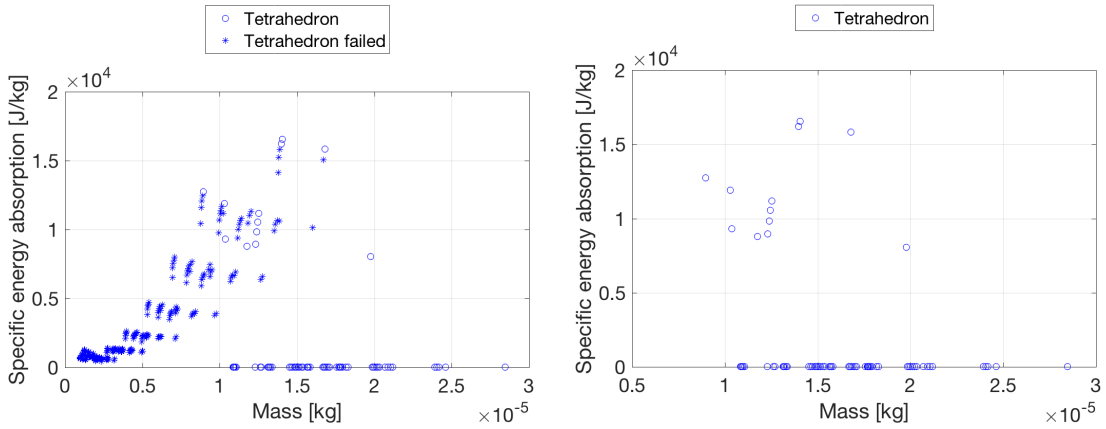


Figure 5.4: Specific energy absorption of tetrahedron initialization. Failed combinations (*) are included in the left image and excluded in the right image.

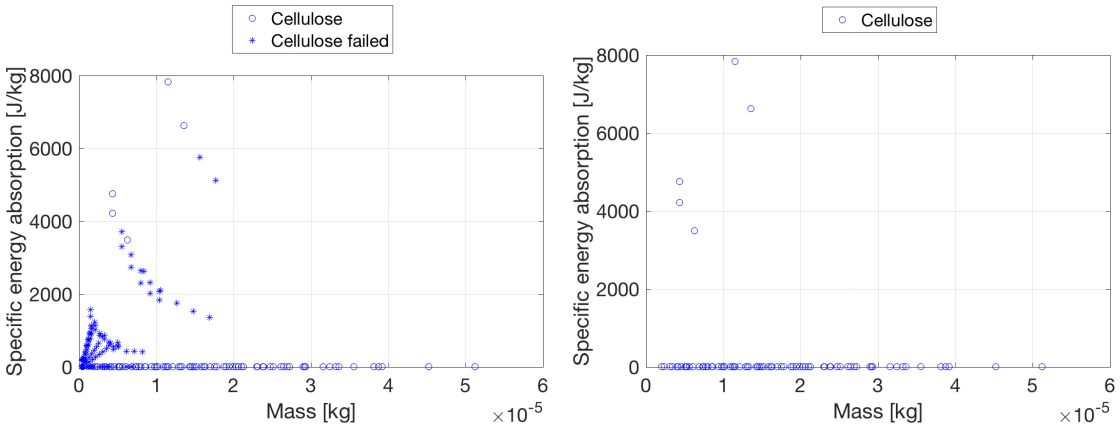


Figure 5.5: Specific energy absorption of cellulose initialization. Failed combinations (*) are included in the left image and excluded in the right image.

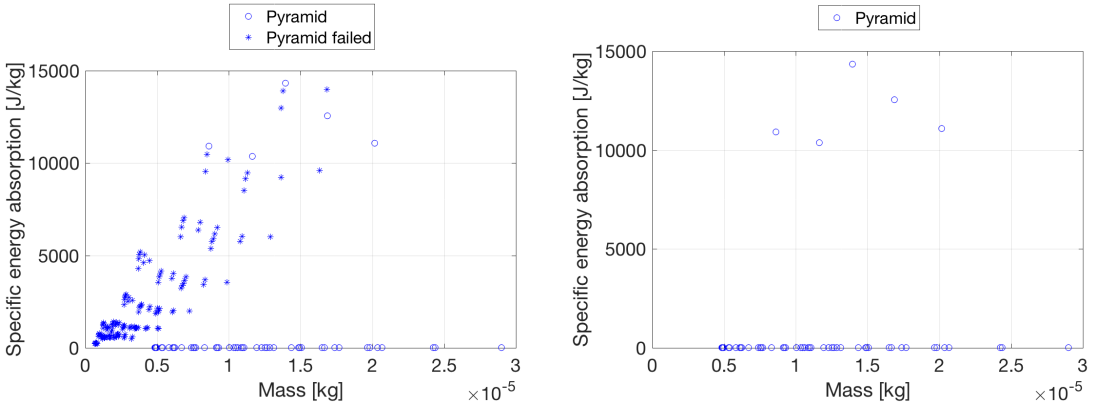


Figure 5.6: Specific energy absorption of pyramid initialization. Failed combinations (*) are included in the left image and excluded in the right image.

5.4 Optimization algorithm

The results from the initialization defines the start guess for the optimization algorithm. All combinations performed from the initialization are presented as specific energy absorption vs beam radius (Figure 5.7), number of unit cells (Figure 5.8), width (Figure 5.9) or height factor (Figure 5.10). In these figures, failed combinations are marked with asterisk (*). The figures are related to the cellulose structure, but similar graphs are generated for the tetrahedron and pyramid as well, see Appendix A. The design variables display areas of higher specific energy absorption. In the case of varying radius, Figure 5.7, the result varies from failed combinations to too stiff combinations as the beams varies from thin to thick. There seems to be a threshold around 1.5 - 1.7 mm in radius, where the structure is stiff enough to hold as well as absorb a significant amount of energy.

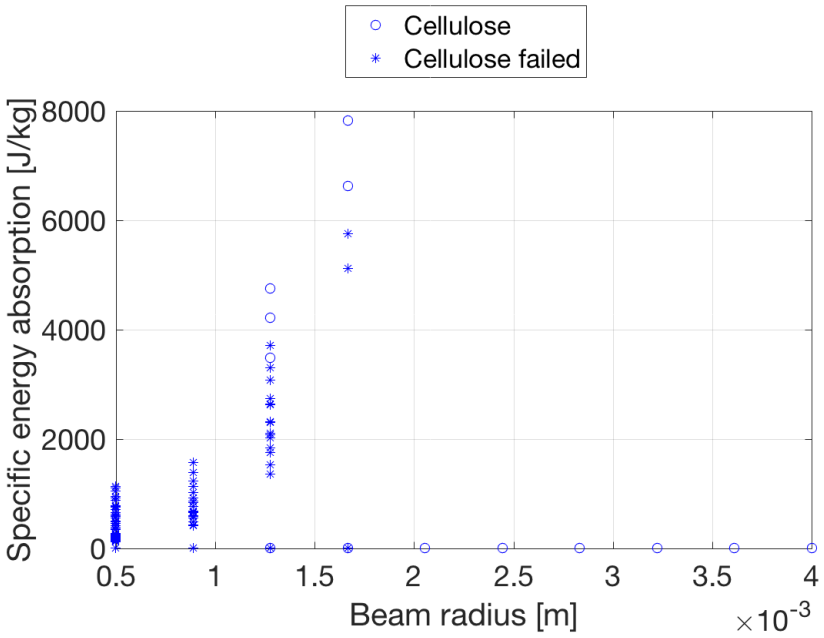


Figure 5.7: Specific energy absorption for cellulose from rough mesh vs beam radius.

In a rough mesh, the number of unit cells are less thus capturing little detail as shown in Figure 5.8. At a large amount of unit cells, large number of beams result in narrow space. Since the maximum radius is dynamically evaluated, large values will not be tested since they collide with each other making an impossible geometry. Hence only small radius are tested, resulting in failed combinations. At one unit cell, the structure either buckles or is absorbs energy only elastically. It is tempting to evaluate the number of unit cells with a finer resolution to display in greater detail what happens between 2 and 6 unit cells. However, the computational time for the initial rough mesh is very sensitive to the number of start tests and needs to be prioritized. The radius contributes the most to the specific energy absorption, hence has the most start tests.

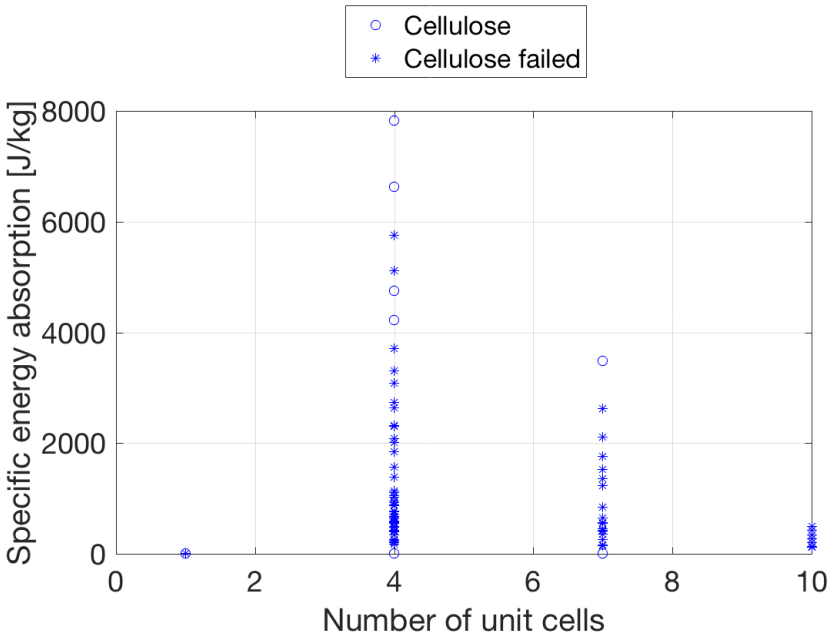


Figure 5.8: Specific energy absorption for cellulose from rough mesh vs number of unit cells.

As the unit cell width decreases, the specific energy absorption associated increases. The width is the only design variable affecting the applied load. Greater the area, greater is the applied force in order to remain the applied stresses, see Section 4.3. The force is evenly applied on the top nodes, which remains the same amount regardless of the width. In the case of cellulose, with increased width the vertical beam lengths remains unchanged. This results in an unchanged critical buckling load, but an increase in applied load. For constant remaining design variables, with increasing width there will be three stages itemized below, see Figure 5.9. At a width of 27 mm, all combinations have reached the third stage in varying width.

- Small width equals small applied load and no unit cell will buckle. This results in a small amount of energy absorption.
- As the width increases, the applied load increases as well. The width increases the area, thus also the force, as a square function. As the applied force increases, unit cells starts to buckle and increases the energy absorption.
- With even longer width, the force will eventually overcome all beams critical buckling load and buckle all unit cells.

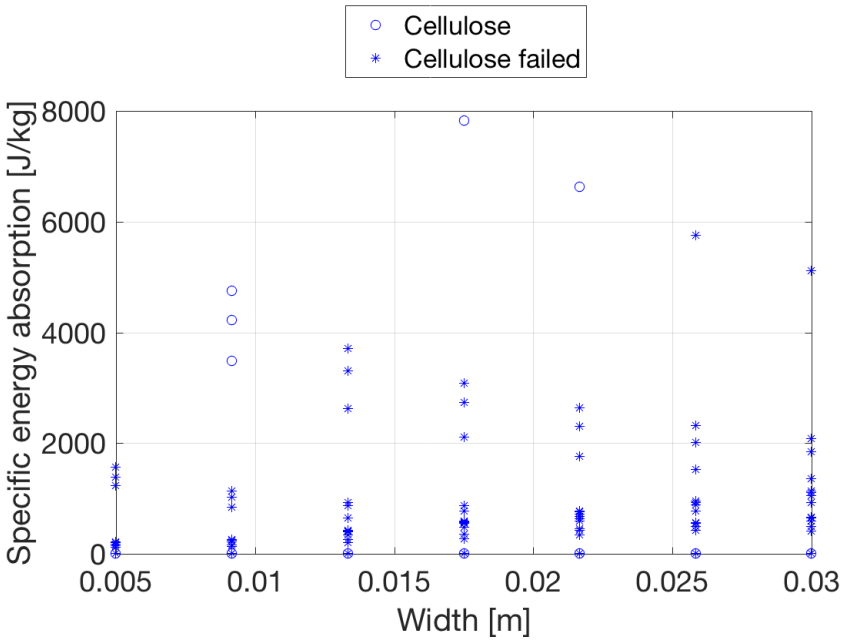


Figure 5.9: Specific energy absorption for cellulose from rough mesh vs unit cell width.

As discussed in Section 5.2, the height factor has little influence on the mass of the system. However, the beam length depends on the height factor, as is the critical buckling load. With increased height factor, the height for the lowest unit cell decreases. When the lowest unit cell is too short, only combinations with very narrow beams will be generated. With short lowest unit cell, entails higher unit cells for the other layers, especially with great height factor. This results in long beams for the upper unit cells that have a small radius, providing a small critical buckling load. Hence, a large proportion of the combinations fails with increasing height factor, see Figure 5.10.

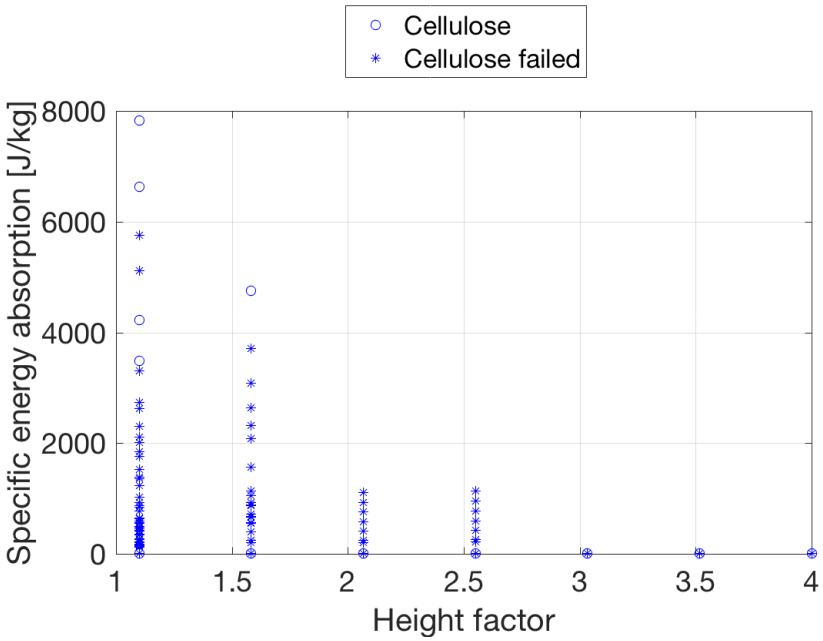


Figure 5.10: Specific energy absorption for cellulose from rough mesh vs height factor.

Each design variable has a value of which the specific energy absorption is the greatest. This is stored as the start guess for the fine tuning optimization algorithm. The start guesses for each geometry is summarized in Table 5.3. The values for cellulose are from Figure 5.7, 5.8, 5.9 and 5.10.

Table 5.3: Start guess from the rough combination test.

	Tetrahedron	Cellulose	Pyramid
Number of unit cells	3	4	3
Beam radius [mm]	1.65	1.65	1.65
Width [mm]	30.0	16.5	30.0
Height factor	2.0	1.1	2.0
Mass [g]	0.0138	0.0108	0.0137
Energy absorption [J]	0.2196	0.0846	0.1884
Specific energy absorption [kJ/kg]	15.9	7.8	13.8

5.4.1 Refinement of design variables

With the configuration of design variable generating the highest specific energy absorption as a start guess, we adjust one variable at a time. This evaluation clarifies the behaviour of the energy absorption depending on each parameter. Every time a new highest specific energy is identified, the start guess is updated and the script reruns. The characteristics for varying each parameter remains the same and will be evaluated separately, starting with the most influential variable; the radius. With the remaining design variables fixed, the results on specific energy absorption as the radius varies is displayed in Figure 5.11. Four stages are identified.

- Small beam radius ($r < 1.80$ mm) buckles all unit cells, hence not being able to withstand the applied force, see section 4.3.
- Beam radius between 1.80 mm and 2.05 mm buckles all but the last unit cell, generating maximum specific energy absorption.
- Beam radius at 2.05 mm buckles only the top unit cell. The structure is highly sensitive to the radius around this point. Small deviations in radius results in either 1, 2 or 0 buckled unit cells.
- Greater beam radius than 2.05 mm provides a too stiff structure and no layer will buckle.

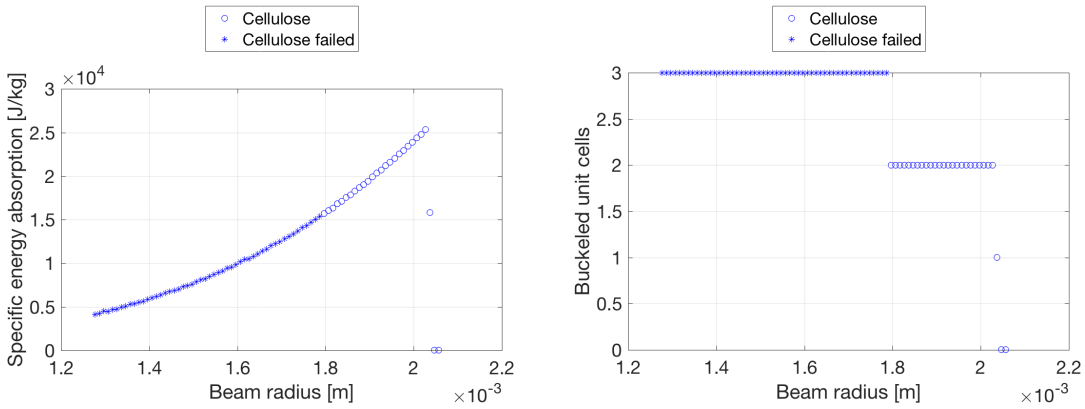


Figure 5.11: Specific energy absorption (left image) and buckled unit cells (right image) depending on beam radius.

Note that the radius is only tested within the span from the rough mesh where the start guess is located and its closest neighbors, see Figure 5.7. Furthermore, the number of unit cells are tested with the updated start guess. With higher beam radius, it is found that 3 unit cells generates the highest specific energy absorption instead of 4 as the initial start guess implies, Figure 5.12.

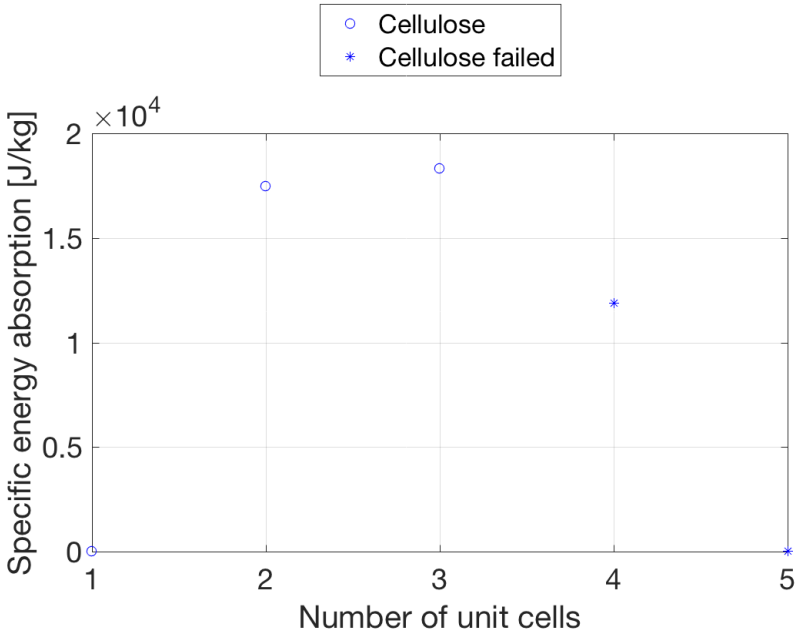


Figure 5.12: Specific energy absorption dependent on number of unit cells for the cellulose geometry.

Following the same procedure, testing the width of a unit cell and the height factor individually with the other parameters fixed, the results in Figure 5.13 are found. These results however vary from the number of unit cells and beam radius since no local maximum is found within the design range. This is due to an updated start guess that deviates from the initial one. The number of unit cells are 3 instead of 4, meaning that all rough combinations with 3 unit cells are missed in the rough mesh. This change might have another local maximum for specific energy absorption outside the design range for the refinement. However, the script cannot adjust for these behaviours. A solution would be to run the initial rough mesh with finer resolution on the number of unit cells, on the cost of computational time, see Section 4.5.2.

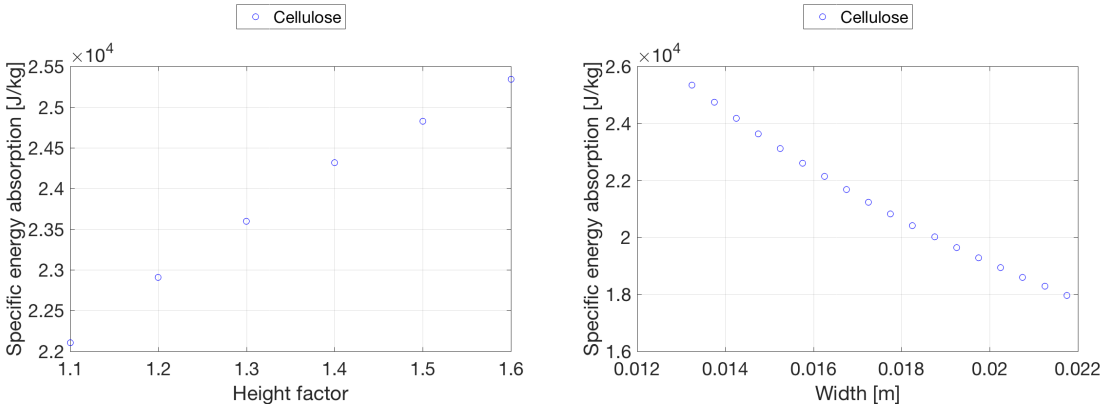


Figure 5.13: Specific energy absorption and height factor (left image) or unit cell width (right image).

All successful tests, both from the rough mesh and the refinement, are gathered in Figure 5.14. The diverging scattered tests are rough mesh and by varying the number of unit cells. The gathered tests, forming lines of dots in the figure are results from the refinement when variations for each design variable is performed.

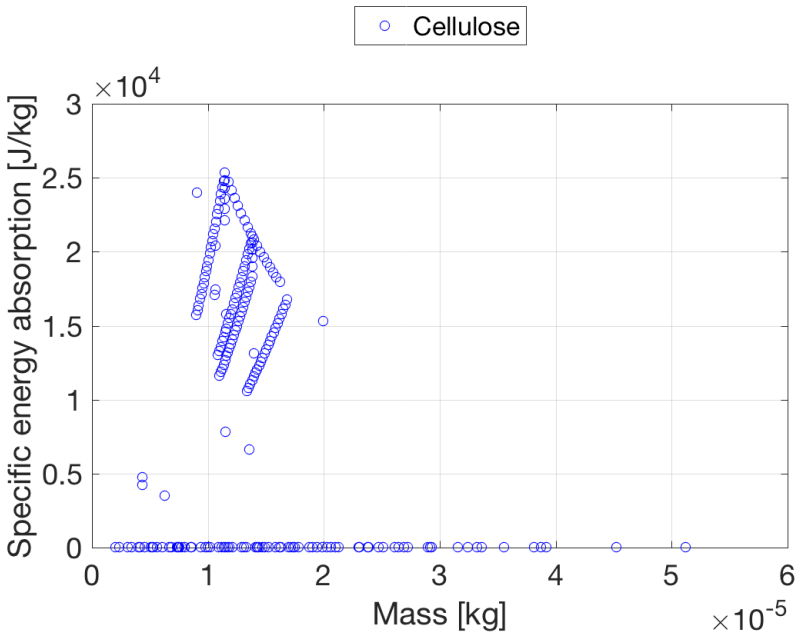


Figure 5.14: Specific energy absorption for all successful tests for the cellulose structure.

5.5 Final geometries and their ranking

The final solution for the three geometries are presented in Table 5.4. Ranking by specific energy absorption, the cellulose is best followed by the tetrahedron and lastly the pyramid. By looking at energy absorption, both the tetrahedron and cellulose have similar values. However, since the cellulose has fewer beams in its unit cell, the mass of the cellulose is generally lower than the tetrahedron. Even if the radius for the cellulose is larger, the total mass of the structure is lower than for both tetrahedron and pyramid. This results in a higher specific energy absorption than for the other structures. The final geometries are displayed as 3D plots in Figure 5.15, 5.16 and 5.17 both as the output from the script and CAD models. The CAD models are depicted as four by four unit columns to demonstrate the structures repeatability. The CAD model is constructed from the final solution for each geometry, including the beam radius which cannot be displayed in the Matlab plot. These structures represents the unit columns that are build up of finite element beams and are tested in the program. The beam radius is not represented in the graph. Each unit column has the symmetry boundary condition and will be repeated in x- and y-direction if manufactured. The difference in complexity between the structures are clearly displayed in the graphs, as the tetrahedron geometry looks very dense compared to the other.

Table 5.4: Optimized design parameters and final result.

	Tetrahedron	Cellulose	Pyramid
Number of unit cells	3	3	3
Beam radius [mm]	1.74	2.03	2.11
Width [mm]	24.75	13.25	19.50
Height factor	1.99	1.60	2.11
Mass [g]	0.014	0.011	0.012
Energy absorption [J]	0.286	0.291	0.245
Specific energy absorption [kJ/kg]	21.088	25.343	20.936

Both the tetrahedron and pyramid have similar properties, even though the design variables are slightly varying. Both are built on the same principle; a pyramid (with either a square or triangular base) with every other layer up-side-down. The connection points in these structures are only placed at beams ends, with multiple beams connecting with one another. The cellulose however has connections in the middle of the horizontally placed beams, creating bending in these beams. The general structure of cellulose is less stiff due to this setup, hence making the normal forces in the beams smaller and able to take on a higher applied force without buckling. This can in turn generate a lighter structure with the same energy absorbing properties as the others. Although, the simulations are only performed for compression tests, it is necessary to evaluate the shear forces and stiffness in the structures. The sought application is to replace the foam in bicycle helmets and in case of a crash the forces is not always perfectly aligned with the normal of the helmet. The tetrahedron and pyramid should be able to withstand shear forces the best since they have diagonal beams in multiple directions. The density of beams in both tetrahedron and pyramid may be of advantage since they create a somewhat chaotic pattern, similar to foam, enabling force paths along any beam. In case of a shear load for the cellulose however will bend the vertical beams easily since there is no diagonal bracing, see Figure 5.16.

Manufacturing of the structures might prove to be difficult, regardless of the selection of geometry. 3D printers struggles with horizontal beams and slender structures, thus needing support structure in order to manufacture the piece. The support structure on the other hand needs to be removed, which will be very difficult with geometries of these scales. The easiest structure to print would be the pyramid since it does not have a large amount of horizontal beams and the existing ones are connected with multiple diagonal beams. The pyramid is not as dense as the other structures, making removal of eventual support material easier. The pyramid also has the largest beam radius, providing a good support for the structure during manufacturing. The cellulose is quite narrow and have multiple horizontal beams making it hard to manufacture, even if the structure is self is spacious. The tetrahedron, as mentioned above, has a greater amount of beams in its structure, making space more narrow. Each unit cell in the tetrahedron has twelve horizontal beams, of which four are combined with the unit cell below and four with the one above. The radius for the tetrahedron is smaller than the other structures, making it more difficult to manufacture and more support material is needed.

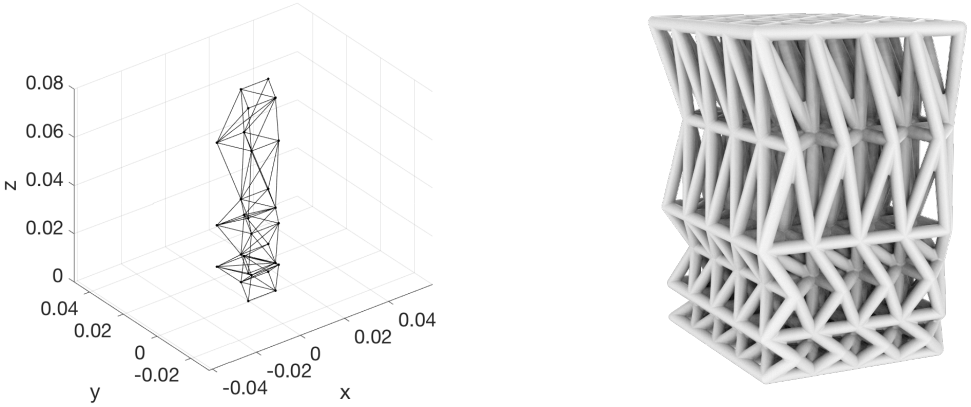


Figure 5.15: Final geometry of tetrahedron as one unit column (left image) and 4 by 4 unit columns (right image).

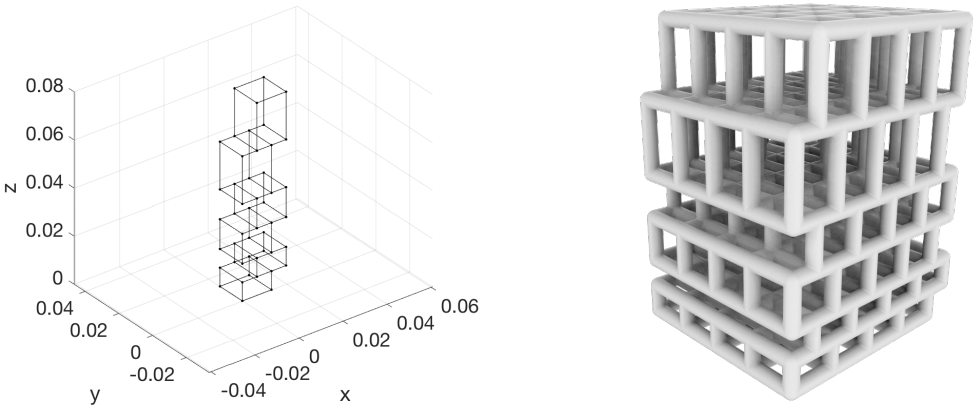


Figure 5.16: Final geometry of cellulose as one unit column (left image) and 4 by 4 unit columns (right image).

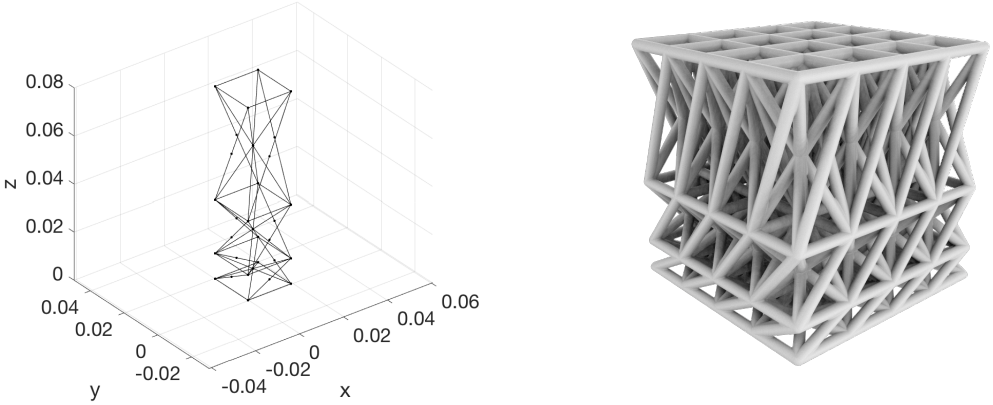


Figure 5.17: Final geometry of pyramid as one unit column (left image) and 4 by 4 unit columns (right image).

5.5.1 Performance of the optimization procedure

The performance of the optimization procedure is evaluated for how many combinations that are tested against all unique combinations. In Figure 5.18 the calculated cases, from Section 4.5.2, are displayed as percentage of all combinations. The dotted lines represent the actual setup, Table 5.1, and the solid lines are references calculated if all design variables had the same resolution, i.e. number of tests from 1 to 10. The best and worst case scenario is the outcome of optimization algorithms. There is potential for a faster calculation time by reducing the resolution for the rough mesh. However, this increases the risk of missing sought configurations and a good start guess, see example in Section 5.4.1.

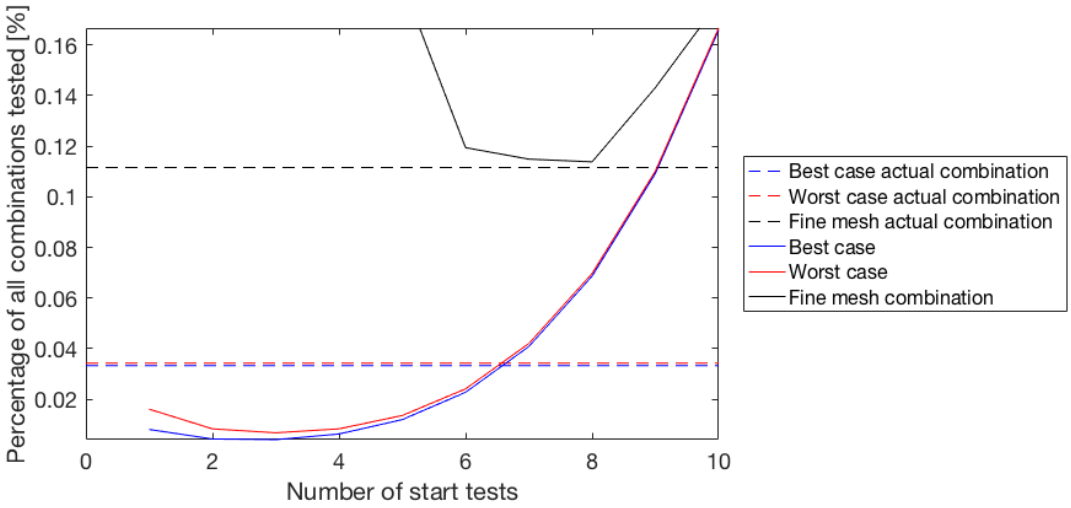


Figure 5.18: Number of tests performed for the cellulose structure as percentage of all unique combinations.

The black lines represents an alternative to the optimization algorithm for the fine tuning, Figure 5.18. This method simply tests all unique combinations in the design space surrounding the initial start guess. This method will take significantly more tests, hence increasing the computational time. However, it guaranties that all combinations within the design range is tested. A summary for all structures performance with their respective setup are presented in Table 5.5. As Figure 5.18 implies, a more coarse rough mesh will generate fewer tests with the compromise of a worse start guess. For example, the cellulose setup will generate 6 million combinations while the tetrahedron setup generates 17 million. However, because the cellulose is more sensitive to variation in the design variables, a finer rough mesh is needed to initialize and find a good start guess. The rough mesh is therefore 1,960 tests for the cellulose but only 450 for the tetrahedron. The final number of tests that were done for the cellulose and tetrahedron is 1,074 and 669, respectively. As seen in the table, both setups for the cellulose and pyramid results in fewer tests being performed than the best case scenario. This is due to the dynamic upper limit for the beam radius. Before every test is performed, the active combination of design variables are evaluated to determine if they will generate a possible structure. If that particular combination has e.g. a large beam radius and a small unit

cell height generating colliding beams, that combination is not added to the tests. Hence the actual tests performed may be smaller than the best case scenario. Note that even with this optimization method, a large amount of tests will be performed. This thesis approximates the problem into an elastic model with beam buckling, individually evaluated. With this, the solution time varies up to 1.5 hours when each test takes 5 seconds to perform. If a conventional software were to be used with implicit, explicit or response spectrum the solution time for a single test will take hours to perform. A FEM model with plasticity would also significantly increase the computational time and not making the quantities of combinations possible.

Table 5.5: The performance of the script.

	Tetrahedron	Cellulose	Pyramid
Total number of combinations	17391123	6058800	481572
Rough mesh combinations	450	1960	450
Fine combinations	42258	6760	1570
Worst case	537	2075	486
Best case	670	2016	525
Actual tests performed	669	1074	429
Percentage of all combinations tested	0.004%	0.018%	0.089%
Total computation time [minutes]	56	90	36

6
Conclusion

The script successfully generates sought geometries and autonomously updates design variables in order to find values for the variables generating maximum specific energy absorption. The method for finding these values are effective and only tests a small proportion of the total number of unique combination of design variables. The model is easily expandable, should the need arise to extend with another geometry or adjust the material properties. The result is highly dependent on a good start guess and design variable range and resolution. Multiple analysis are recommended with adjusted start setup to ensure that the highest specific energy absorption is found.

Through the results, the design variable with most influence on specific energy absorption is concluded: the beam radius. Of the three geometries cellulose, tetrahedron and pyramid studied, cellulose generates the highest specific energy absorption of 25 kJ/kg. The design values providing maximum specific energy absorption for cellulose are: number of unit cells 3, beam radius 2.03 mm, width 13.25 mm and a height factor of 1.60. The maximum value are within the range of manufacturing capabilities, hence physical tests can be performed for validation. The conceptual difference between cellulose and tetrahedron or pyramid is how the force paths are constructed. In tetrahedron and pyramid structure, all beams are connected through nodes at the beams endpoints resulting in only tension or compression in the beams. The beams in the cellulose structure however will create bending in the horizontal beams, resulting in a less stiff structure. Based on this study, beams in bending absorbs higher energy than beams in compression or tension. Although cellulose has the highest energy absorption, the specific energy absorption increases further since the mass is lower for cellulose than the other structures.

Based on the results and the discussion on manufacturing, see Section 5.5, cellulose is the best structure among the three studied in this thesis. Cellulose has both high energy absorption and specific energy absorption and is manufacturable. Its characteristic design should generate a smooth deceleration and provide a good substitute to foams in helmets.

6.1 Future work and recommendations

This thesis is meant to explore the field of custom made structures for energy absorption. Future work can focus either in making the model more accurate, more efficient, more flexible or exploring more types of structures or materials. The accuracy of the model can be improved by performing physical compression tests with the final geometries provided by the script. From this evaluation, the model can be adjusted to better reflect the reality. A plastic model should improve the result accuracy, on the cost of performance. It is recommended to further investigate and implement a plastic FEM model into the script. The efficiency could be improved by e.g. implementing a Newton forward method to the fine-tuning algorithm instead of increment the design variables with a predefined step size. The flexibility of the model can be extended by adding additional design variables, e.g. a radius factor. A varying radius, similar to the height factor, could be implemented to resemble the varying radius found in nature. This would result in a decreasing or increasing radius for each unit cell in the structure. The model is constructed to easily add additional geometries. Example of geometries could be unit cells similar to a hexagonal, cube or asymmetric pyramids or other polyhedrons.
Furthermore, the Matlab script could be combined with CATIA to autonomously generate the structures if an explicit simulation ought to be performed in ANSYS. If a crash test is to be simulated, it is recommended to perform explicit analysis in ANSYS. This procedure would generate accurate crash test results but is very time consuming. For this to be possible, it is recommended to do a more sophisticated optimization algorithm to reduce the number of tests and run the whole analysis on a computer cluster.

Bibliography

[1] National Academy of Sciences. Review of Department of Defense Test Protocols for Combat Helmets. 2014 March 31.

[2] trafikverket.se. Bicycle helmet. Updated 2015 March 10. Available from trafikverket.se

[3] DC Thompson, F Rivara, R Thompson. Helmets for Preventing Head and Facial Injuries in Bicyclists. 1999 October 25. DOI 10.1002/14651858.CD001855.

[4] helmets.org. Bicycle Helmets Liners: Foam and Other Materials. Updated 2016 December 30. Available from helmets.org

[5] Cited 17 May 2017. Available from https://www.bikerumor.com/2010/06/12/bikerumor-exclusive-review-kali-avita-carbon-xc-am-helme/

[6] Luca DL, Giuseppe S, Daniela O. Deformation Mechanisms and Energy Absorption of Polystyrene Foams for Protective Helmets. Elsevier. 2001 Apr 25; Polymer Testing 21 (2002) 217-228

[7] study.com. Compact Bone: Definition, Structure & Function. Updated 2017. Available from study.com/academy/lesson/compact-bone-definition-structure-function

[8] innerbody.com. Cortical (Compact) Bone. Updated 2017. Available from innerbody.com/image_skel09/skel61

[9] Cited 17 May 2017. Available from http://teachmeanatomy.info/wp-content/uploads/Structure-of-Mature-Bone.jpg

[10] study.com. Cancellous Bone: Definition, Structure & Function. Updated 2017. Available from: study.com/academy/lesson/cancellous-bone-definition-structure-function

[11] Cited 17 May 2017. Available from http://study.com/academy/lesson/trabeculae-of-bone-definition-function.html

[12] study.com. Trabeculae of Bone: Definition, Structure & Function. Updated 2017. Available from study.com/academy/lesson/trabeculae-of-bone-definition-function

[13] P Zaslansky, A.A Friesem, S Weiner. Structure and Mechanical Properties of the Soft Zone Separating Bulk Dentin and Enamel in Crowns of Human Teeth: Insight Into Tooth Function. 2005 October 12. Journal of Structural Biology. 153(2):188-99.

[14] Cited 17 May 2017. Available from https://en.wikipedia.org/wiki/Crown_(tooth)#/media/File:Blausen_0863_ToothAnatomy_02.png

[15] I Bjurhager. Effects of Cell Wall Structure on Tensile Properties of Hardwood. 2011 Apr. Available from: KTH School of Chemistry

[16] D.J Cosgrove. Growth of the plant cell wall. Volume 6 2005 Nov. Available from: www.nature.com
[17] E Badel, F.W Ewers, H Cochard, F.W. Telewski. Acclimation of mechanical and hydraulic functions in trees: impact of the thigmomorphogenetic process. 2015 Apr. Available from: National Center for Biotechnology Information
[18] G Lu, T.X Yu. Energy Absorption of Structures and Materials. Woodhead Publishing Limited: Cambridge England; 2003.
[19] European Committee for Standardization. Helmets for Pedal Cyclists and for Users of Skateboards and Roller Skates. 1997 February. Ref. No. EN 1078: 1997 E. ICS 13.340.20.
[20] j Higgins. Drop Test Simulation Made Easy With Ansys Simulation. Ansys Inc. 2008.
[21] Dan Zenkert. An introduction to Sandwich Structures - student edition. 2:nd Edition 2005 Stockholm.
[22] N Ottosen, H Petersson. Introduction to the FINITE ELEMENT METHOD. 1992.
[23] L Andersen, S.R.K. Nielsen. Elastic Beams in Three Dimensions. DCE Lecture Notes No.23. Aalborg University, Department of Civil Engineering - Structural Mechanics. August 2008
[24] P-E Austell, O Dahlblom, J Lindemann, A Olsson, K-G Olsson, K Persson, H Petersson, M Ristinmaa, G Sandberg, P-A Wernberg. CALFEM, a Finite Element Toolbox. Version 3.4. Lund Universiity; 2004.
[25] Cited 17 May 2017. Available from http://www.ck12.org/life-science/Plant-Cell-Structures-in-Life-Science/lesson/Plant-Cell-Structures-MS-LS/
[26] mathworld.wolfram.com. E.W Weisstein. "Tetrahedron." Cited 19 April 2017. Available from: http://mathworld.wolfram.com/Tetrahedron.html
[27] mathworld.wolfram.com. E.W Weisstein. "Pyramid.". Cited 19 April 2017. Available from: http://mathworld.wolfram.com/Pyramid.html
[28] E. Svensson. Material Characterization of 3D-Printed Energy-Absorbent Polymers Inspired by Nature. 2017. Materials and Manufacturing Technology, Chalmers.
[29] N. J. Mills. Protective Capability of Bicycle Helmets. Butterworth & Co Ltd. 1990. 0306-3674/90/010055-06.
[30] silver.neep.wisc.edu. Materials with Strucural Hierarchy. Updated 1993. Available from silver.neep.wisc.edu
[31] R Hill, 1963. Elastic properties of reinforced solids: some theoretical principles. J. Mech. Phys. Solids 11 (5), 357–372.

A

Appendix 1 - Result

A.1 Tetrahedron result

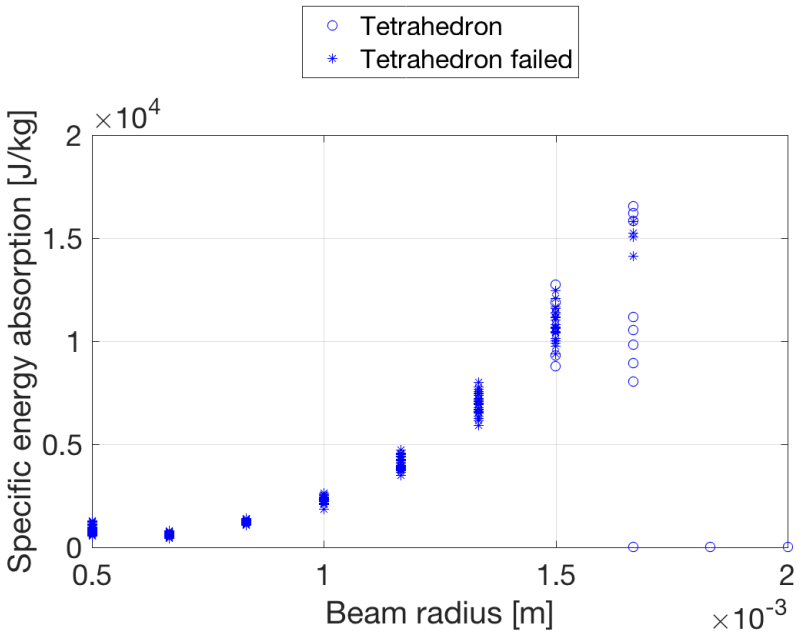


Figure A.1: Specific energy absorption for tetrahedron from rough mesh vs beam radius.

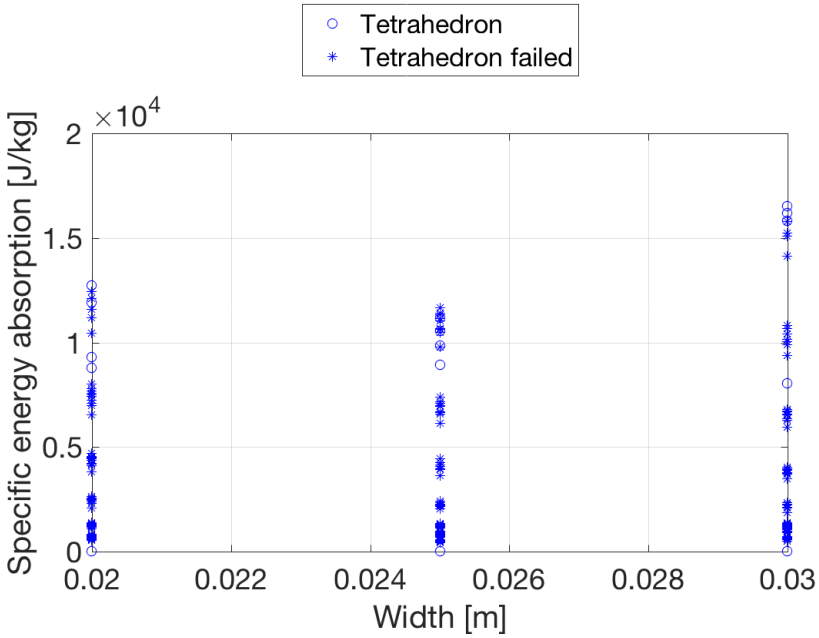


Figure A.2: Specific energy absorption for tetrahedron from rough mesh vs unit cell width.

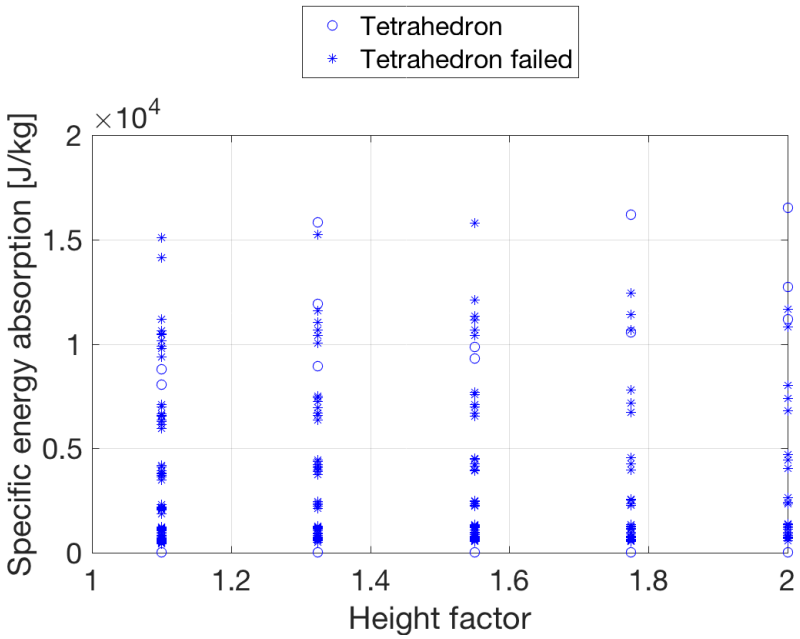


Figure A.3: Specific energy absorption for tetrahedron from rough mesh vs height factor.

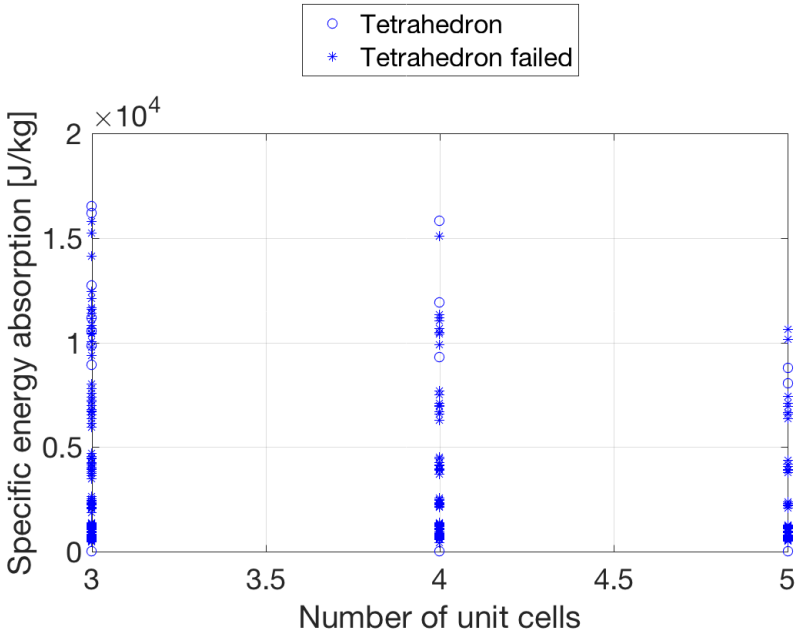


Figure A.4: Specific energy absorption for tetrahedron from rough mesh vs number of unit cells.

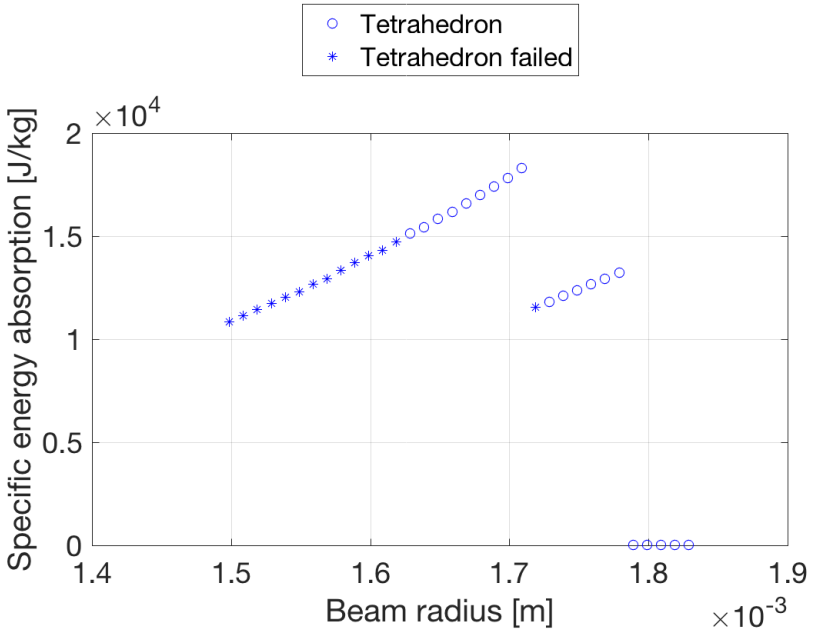


Figure A.5: Specific energy absorption for tetrahedron with varying beam radius.

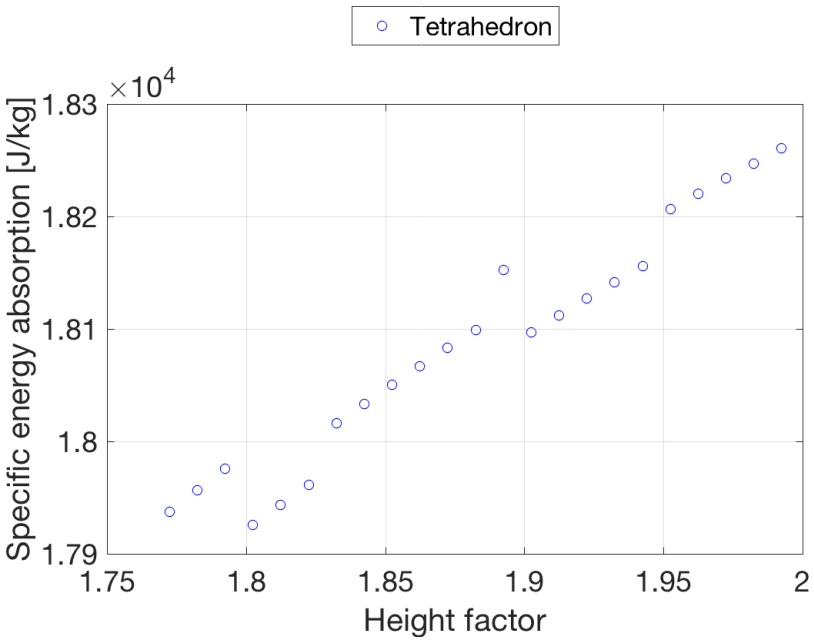


Figure A.6: Specific energy absorption for tetrahedron with varying height factor.

A.2 Pyramid result

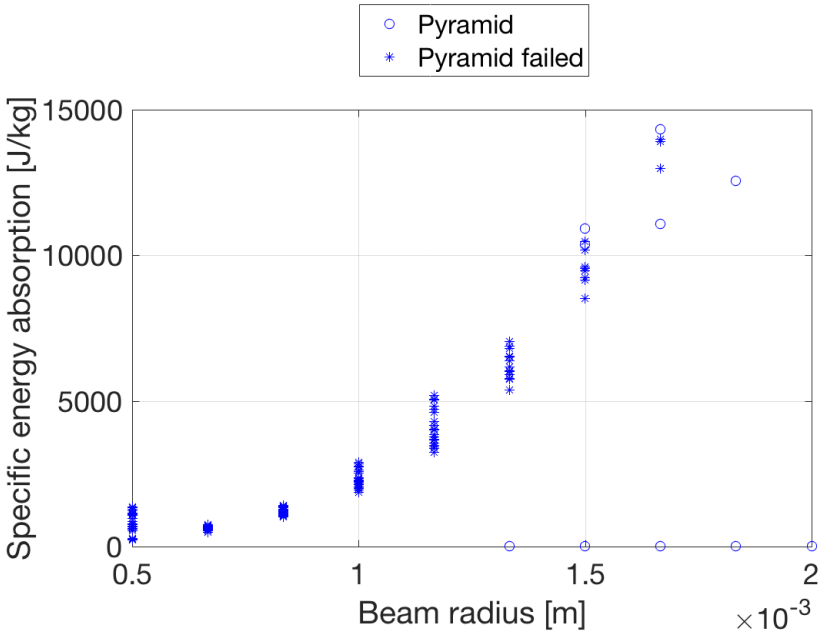


Figure A.7: Specific energy absorption for pyramid from rough mesh vs beam radius.

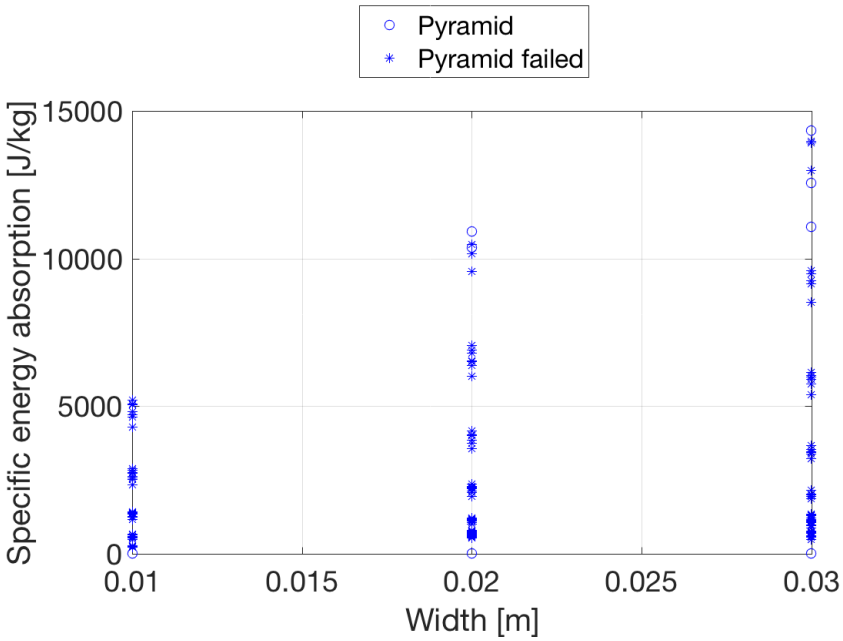


Figure A.8: Specific energy absorption for pyramid from rough mesh vs unit cell width.

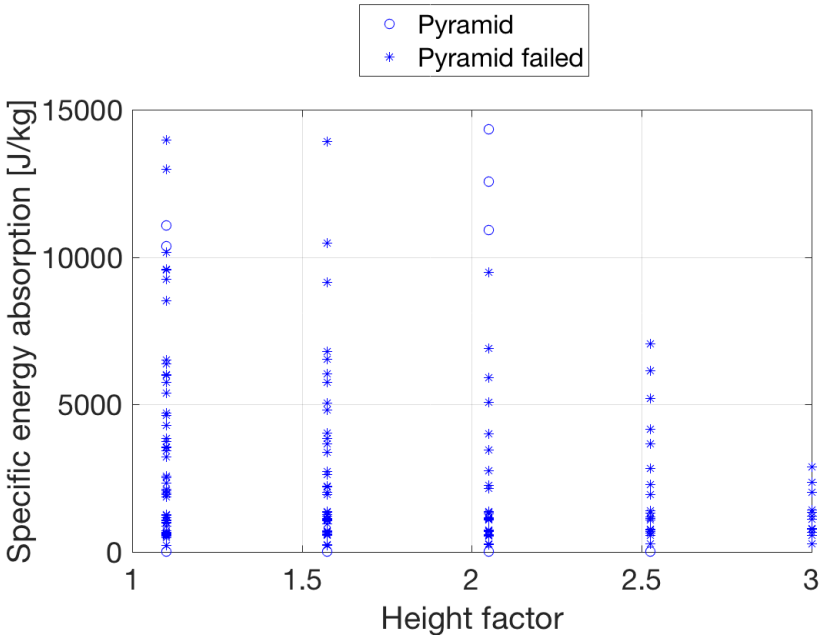


Figure A.9: Specific energy absorption for pyramid from rough mesh vs height factor.

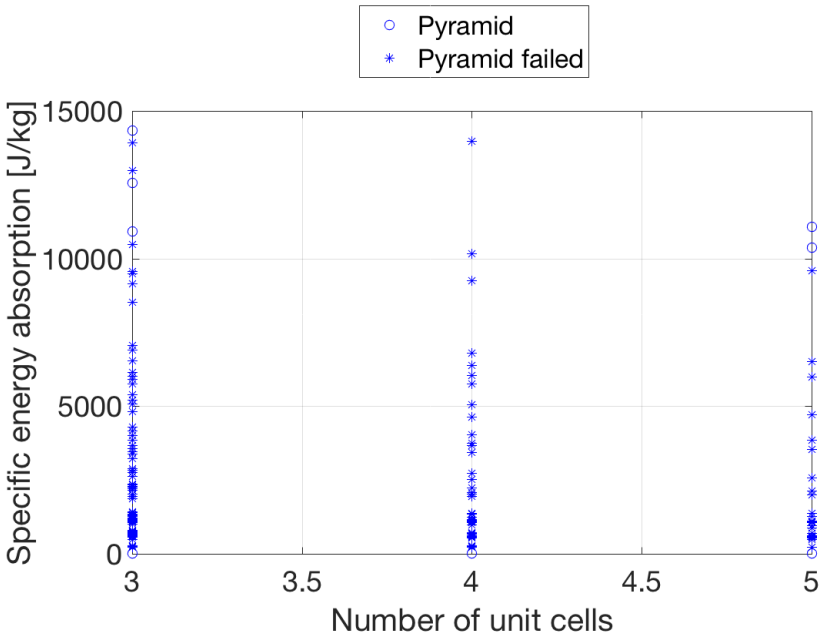


Figure A.10: Specific energy absorption for pyramid from rough mesh vs number of unit cells.

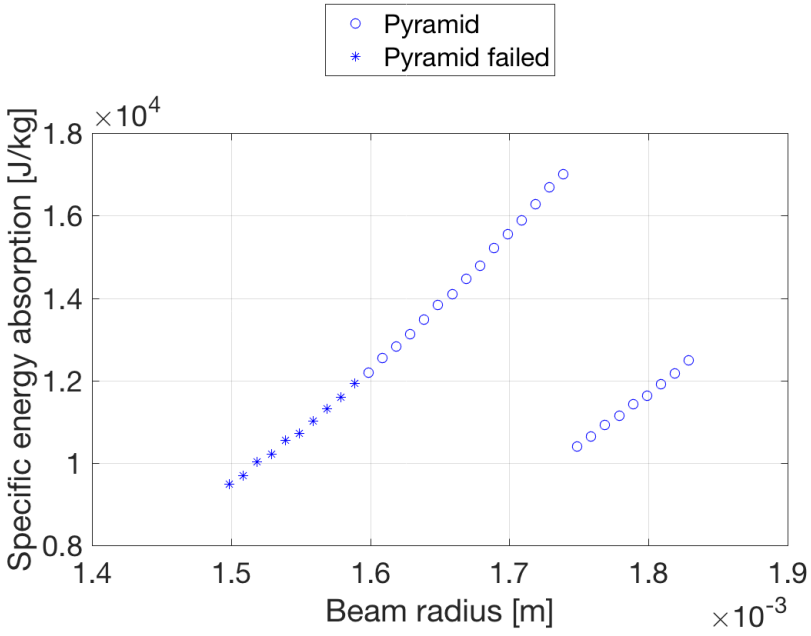


Figure A.11: Specific energy absorption for pyramid with varying beam radius.

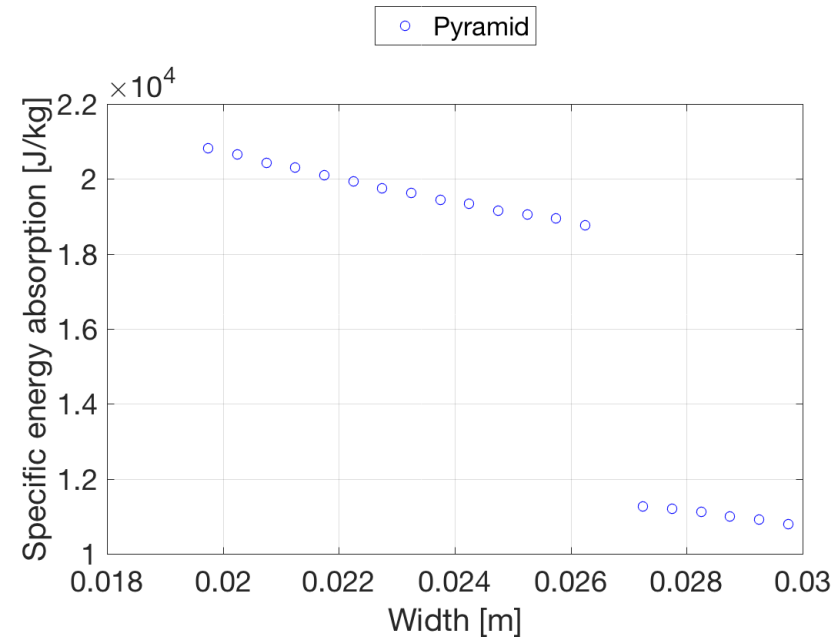


Figure A.12: Specific energy absorption for pyramid with varying width.

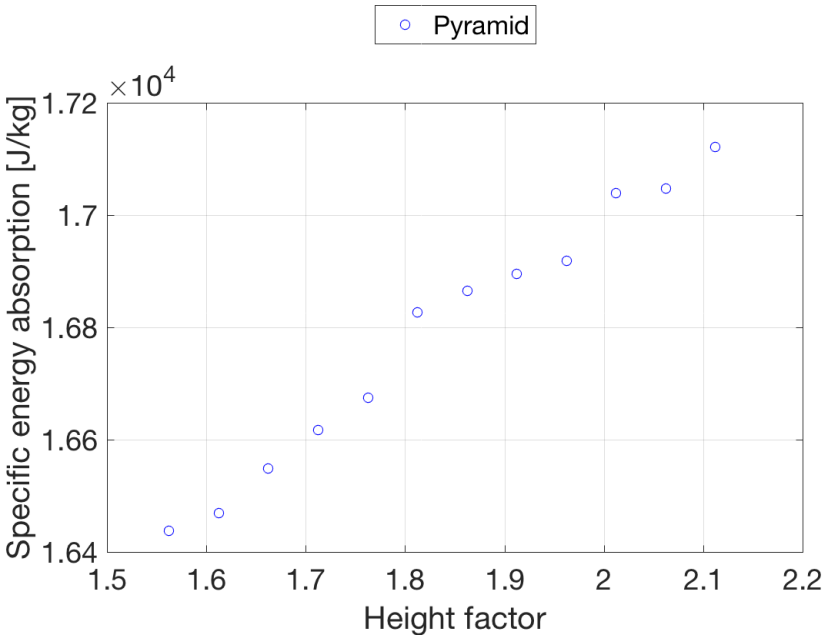


Figure A.13: Specific energy absorption for pyramid with varying height factor.

B

Appendix 2 - Matlab code

B.1 Main

```
%-------------------------------------------------------------------------
% PURPOSE
% Optimize a selected geometry within the prescribed design range.
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

clc
close all
clear variables

% Global variables
global mainData E G rho sigma_y height plotMode testStructure force ...
    startGuess

% Test variables
testStructure = 3;
    % 1 = Tetrahedron
    % 2 = Cellulose
    % 3 = Pyramid
    % 4 = Cube

% Design variables
    % Number of unit cells
    nrOfLevels.min = 3;
    nrOfLevels.max = 5;
    nrOfLevels.resolution = 1;
    nrOfLevels.startTests = 3;
    nrOfLevels.meshSize = 3;

    % Height factor
    heightFactor.min = 1.1;
    heightFactor.max = 3;
    heightFactor.resolution = 0.05;
    heightFactor.startTests = 5;
    heightFactor.meshSize = ((heightFactor.max - heightFactor.min + ...
        heightFactor.resolution) / heightFactor.resolution) ...
        / (heightFactor.startTests - 1);
```

```
40
41      % Beam radius
42      beamRadius.min = 0.5e-3;
43      beamRadius.max = 2.0e-3;
44      beamRadius.resolution = 0.01e-3;
45      beamRadius.startTests = 10;
46      beamRadius.meshSize = ((beamRadius.max - beamRadius.min + ...
47          beamRadius.resolution) / beamRadius.resolution)...
48          / (beamRadius.startTests - 1);
49
50      % Side width
51      sideWidth.min = 10e-3;
52      sideWidth.max = 30e-3;
53      sideWidth.resolution = 0.5e-3;
54      sideWidth.startTests = 3;
55      sideWidth.meshSize = ((sideWidth.max - sideWidth.min + ...
56          sideWidth.resolution) / sideWidth.resolution) ...
57          / (sideWidth.startTests - 1);
58
59  % Constraints and defined data
60  height = 80e-3;       % Total height [m]
61  E = 0.94e9;           % Young's modulus [Pa]
62  G = 0.3597e9;         % Shear modulus [Pa]
63  rho = 1160;           % Density [kg/m^3]
64  sigma_y = 31e6;       % Yield strength [Pa]
65  tolerance = 6.25e-6;  % The 3D printers printing accuracy [m]
66
67  % Test setup parameters
68  impactForce = 2500;
69  helmetRadius = 160e-3;
70  impactArea = pi * (helmetRadius * cos(asin(1 - height / helmetRadius)))
        ^2;
71  force.structureStress = impactForce / impactArea;
72  force.min = 0;
73  force.steps = 200;
74
75  % Plots
76  plotMode.grid = 0;
77  plotMode.results = 0;
78  plotMode.progress = 0;
79  plotMode.acceptFailure = 1;
80  plotMode.figureNr = 10;
81  plotMode.autoPlot = 0;
82
83  % Initialize
84  nrOfTests = length(sideWidth) * length(nrOfLevels) * length(beamRadius)
        * ...
85      length(heightFactor) * length(testStructure);
86  mainData.tetrahedral = zeros(nrOfTests / length(testStructure), 11);
87  mainData.cellulose = zeros(nrOfTests / length(testStructure), 11);
88  mainData.pyramid = zeros(nrOfTests / length(testStructure), 11);
89  mainData.cube = zeros(nrOfTests / length(testStructure), 11);
90
91  % Start guess
92  startGuess.N = 0;
93  startGuess.hf = 0;
```

```
94  startGuess.r = 0;
95  startGuess.w = 0;
96  startGuess.mass = 0;
97  startGuess.force = [];
98  startGuess.displacement = [];
99  startGuess.buckledLevels = 0;
100 startGuess.energyAbsorption = 0;
101 startGuess.specificEnergyAbsorption = 0;
102
103 % Display setup and calculation time
104 fig = NumberOfTestsPlot(nrOfLevels, heightFactor, beamRadius, sideWidth
        , 5);
105 print(fig, 'AutoPlots/NumberOfTests', '-dpng');
106
107 % Run initial test
108 testNr = RoughCombinationTest(nrOfLevels, heightFactor, beamRadius, ...
109     sideWidth);
110
111 % Define mesh size
112 testNr = optimisationAlgorithm(nrOfLevels, heightFactor, beamRadius,
        ...
113     sideWidth, testNr);
114
115 % Write documentation
116 WriteDocumentation(mainData);
117
118 % Print result
119 fprintf('\nOptimization procedure complete. \nNumber of tests: %.0f\n',
        ...
120     testNr);
121 disp(startGuess);
122
123 % Write solution to text file
124 solutionDoc = fopen('AutoPlots/solution.txt', 'w');
125 fprintf(solutionDoc, strcat('Number of tests: %.0f \n N: %.0f \n ', ...
126     ' hf: %.2f \n r: %f [mm] \n w: %f [mm] \n Mass: %f [g] \n ', ...
127     ' buckled unit cells: %.0f \n Energy absorption: %f [J]\n ', ...
128     ' Specific energy absorption: %f [kJ/kg] \n'), testNr, startGuess.N
          , ...
129     startGuess.hf, startGuess.r*1000, startGuess.w*1000, ...
130     startGuess.mass * 1000, ...
131     startGuess.buckledLevels, startGuess.energyAbsorption, ...
132     startGuess.specificEnergyAbsorption/1000);
133 fprintf(solutionDoc, strcat('\n\n Number of unit cells \n Min: %.0f',
        ...
134     '\n Max: %.0f \n Resolution: %.0f \n Start tests: %.0f \n\n', ...
135     ' Height factor \n Min: %.2f', ...
136     '\n Max: %.2f \n Resolution: %.2f \n Start tests: %.0f \n\n', ...
137     ' Beam radius \n Min: %f', ...
138     '\n Max: %f \n Resolution: %f \n Start tests: %.0f \n\n', ...
139     ' Unit cell width \n Min: %f', ...
140     '\n Max: %f \n Resolution: %f \n Start tests: %.0f \n\n'), ...
141     nrOfLevels.min, nrOfLevels.max, nrOfLevels.resolution, ...
142     nrOfLevels.startTests, ...
143     heightFactor.min, heightFactor.max, heightFactor.resolution, ...
144     heightFactor.startTests, ...
```

```
145     beamRadius.min, beamRadius.max, beamRadius.resolution, ...
146     beamRadius.startTests, ...
147     sideWidth.min, sideWidth.max, sideWidth.resolution, ...
148     sideWidth.startTests);
149 fclose(solutionDoc);
150 
151 % Plot all tests
152 plotMode.acceptFailure = 1;
153 fig = PlotFactory(1, 11, 0, 0);
154 print(fig, 'AutoPlots/AllTests/MassAndEnergyAbsorption', '-dpng');
155 
156 fig = PlotFactory(1, 10, 0, 0);
157 print(fig, 'AutoPlots/AllTests/MassAndSpecificEnergyAbsorption', '-dpng
        ');
158 
159 fig = PlotFactory(3, 10, 0, 0);
160 print(fig, 'AutoPlots/AllTests/WidthAndSpecificEnergyAbsorption', '-
        dpng');
161 
162 fig = PlotFactory(4, 10, 0, 0);
163 print(fig, 'AutoPlots/AllTests/NrOfUnitCellsAndSpecificEnergyAbsorption
        ', ...
164     '-dpng');
165 
166 fig = PlotFactory(5, 10, 0, 0);
167 print(fig, 'AutoPlots/AllTests/HfAndSpecificEnergyAbsorption', '-dpng')
        ;
168 
169 fig = PlotFactory(1, 8, 0, 0);
170 print(fig, 'AutoPlots/AllTests/MassAndBuckling', '-dpng');
171 
172 fig = PlotFactory(8, 10, 0, 0);
173 print(fig, 'AutoPlots/AllTests/BucklingAndSpecificEnergyAbsorption',
        ...
174     '-dpng');
175 
176 fig = PlotFactory(9, 10, 0, 0);
177 print(fig, 'AutoPlots/AllTests/RadiusAndSpecificEnergyAbsorption', ...
178     '-dpng');
179 
180 % Plots excluding failing structures
181 plotMode.acceptFailure = 0;
182 fig = PlotFactory(1, 11, 0, 0);
183 print(fig, 'AutoPlots/AllTests/MassAndEnergyAbsorptionNoFailure', ...
184     '-dpng');
185 
186 fig = PlotFactory(1, 10, 0, 0);
187 print(fig, 'AutoPlots/AllTests/MassAndSpecificEnergyAbsorptionNoFailure
        ', ...
188     '-dpng');
189 
190 fig = PlotFactory(3, 10, 0, 0);
191 print(fig, 'AutoPlots/AllTests/
        WidthAndSpecificEnergyAbsorptionNoFailure', ...
192     '-dpng');
193 
```

```
194 fig = PlotFactory(4, 10, 0, 0);
195 print(fig, ...
196     'AutoPlots/AllTests/
           NrOfUnitCellsAndSpecificEnergyAbsorptionNoFailure', ...
197     '-dpng');
198
199 fig = PlotFactory(5, 10, 0, 0);
200 print(fig, 'AutoPlots/AllTests/HfAndSpecificEnergyAbsorptionNoFailure',
        ...
201     '-dpng');
202
203 fig = PlotFactory(1, 8, 0, 0);
204 print(fig, 'AutoPlots/AllTests/MassAndBucklingNoFailure', '-dpng');
205
206 fig = PlotFactory(8, 10, 0, 0);
207 print(fig, ...
208     'AutoPlots/AllTests/BucklingAndSpecificEnergyAbsorptionNoFailure',
           ...
209     '-dpng');
210
211 fig = PlotFactory(9, 10, 0, 0);
212 print(fig, ...
213     'AutoPlots/AllTests/RadiusAndSpecificEnergyAbsorptionNoFailure',
           ...
214     '-dpng');
215
216 % Plot the structure
217 fig = PlotGeometry(testStructure, startGuess.w, startGuess.N,
       startGuess.hf);
218 print(fig, 'AutoPlots/OptimizedStructure', '-dpng');
219
220 % Plot force vs displacement curve
221 fig = figure(plotMode.figureNr + 1);
222 plot(startGuess.displacement, startGuess.force);
223 xlabel('Displacement [m]');
224 ylabel('Force [N]');
225 set(gca, 'fontsize', 18);
226 print(fig, 'AutoPlots/ForceDisplacement', '-dpng');
227
228 close all;
```

B.2 Rough combination

```
1  function testNr = RoughCombinationTest(nrOfLevels, ...
2      heightFactor, beamRadius, sideWidth)
3  %-------------------------------------------------------------------------
4  %   PURPOSE
5  %   Initializing optimization algorithm by a rough mesh displaying
6  %   combinations with high specific energy absorption.
7  %
8  %   INPUT
9  %       nrOfLevels = value          Number of unit cells
10 %       heightFactor = value        Height factor
```

```
%       beamRadius = value          Bean radius [m]
%       sideWidth = value           Unit cell width [m]
%
%   OUTPUT
%       testNr = value              Test identifier number
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

global height testStructure force startGuess plotMode

% Design variables
N = linspace(nrOfLevels.min, nrOfLevels.max, nrOfLevels.startTests);
hf = linspace(heightFactor.min, heightFactor.max, heightFactor.
    startTests);
r = linspace(beamRadius.min, beamRadius.max, beamRadius.startTests);
w = linspace(sideWidth.min, sideWidth.max, sideWidth.startTests);

% Initialize
nrOfTests = length(w) * length(N) * length(r) * length(hf);
EnergyAbsorption = zeros(1, nrOfTests);
EnergyAbsorption_F = zeros(1, nrOfTests);
Mass = zeros(1, nrOfTests);

% Main iteration loop
testCounter = 1;
testNr = 1;

for i_w = 1 : sideWidth.startTests
% Applied force
nrOfTopNodes = 4;
force.max = force.structureStress * w(i_w)^2 / nrOfTopNodes;

for i_hf = 1 : heightFactor.startTests
for i_N = 1 : nrOfLevels.startTests
for i_r = 1 : beamRadius.startTests
    % Print progress
    fprintf('Test %.0f. r: %.1f [mm], w: %.1f [mm], N: %.0f, hf: %1.1f
        \n', ...
        testCounter, r(i_r)*1000, w(i_w)*1000, N(i_N), ...
        hf(i_hf));
    testCounter = testCounter + 1;

    % Set each level height
    levelHeights = DescribeLevelHeight(hf(i_hf), N(i_N), height);

    % Check if smallest unit cell is large enough
    if min(levelHeights) < 6 * r(i_r)
        fprintf('       Impossible geometry. \n');
        break;
    end

    % Do the simulation
    [Mass(testNr), EnergyAbsorption(testNr), EnergyAbsorption_F(testNr)
        , ...
```

```
64          forceLog, displacementLog, sigma, epsilon, buckledLevels] = ...
65          SimulateTest(testStructure, levelHeights, ...
66          w(i_w), r(i_r));
67
68      % Evaluate if the tested structure is better then the saved one
69      specificEnergyAbsorption = EnergyAbsorption_F(testNr) / Mass(testNr
            );
70      if (specificEnergyAbsorption > startGuess.specificEnergyAbsorption)
71
72          % Check if structure withstood the force
73          if (forceLog(end) ~= 0)
74              fprintf('       Start guess updated!\n');
75
76              % Update the start guess
77              startGuess.N = N(i_N);
78              startGuess.hf = hf(i_hf);
79              startGuess.r = r(i_r);
80              startGuess.w = w(i_w);
81              startGuess.sigma = sigma;
82              startGuess.epsilon = epsilon;
83              startGuess.force = forceLog;
84              startGuess.displacement = displacementLog;
85              startGuess.buckledLevels = buckledLevels;
86              startGuess.mass = Mass(testNr);
87              startGuess.energyAbsorption = EnergyAbsorption_F(testNr);
88              startGuess.specificEnergyAbsorption =
                    specificEnergyAbsorption;
89
90              % Write the updated guess
91              disp(startGuess);
92          end
93      end
94
95      % Store data and results
96      StoreData(Mass(testNr), EnergyAbsorption(testNr), w(i_w), N(i_N),
            ...
97          hf(i_hf), forceLog(end), buckledLevels, r(i_r), ...
98          EnergyAbsorption_F(testNr), testNr, testStructure);
99
100     % Print result for active test
101     fprintf('   Energy absorption: %.2f mJ\n', ...
102         EnergyAbsorption_F(testNr) * 1e3);
103     fprintf('   Specific energy absorption: %.2f kJ/kg\n', ...
104         EnergyAbsorption_F(testNr)/Mass(testNr) * 1e-3);
105
106     % Increase test counter
107     testNr = testNr + 1;
108 end
109 end
110 end
111 end
112
113 % Print progress for each geometry case
114 fprintf('   Geometry %1.0f finished calculations. \n', testStructure);
115
116 % Create plots and save them
```

```
117 % Plots including failing structures
118 plotMode.acceptFailure = 1;
119 fig = PlotFactory(1, 11, 0, 0);
120 print(fig, 'AutoPlots/Initialization/MassAndEnergyAbsorption', '-dpng')
    ;
121
122 fig = PlotFactory(1, 10, 0, 0);
123 print(fig, 'AutoPlots/Initialization/MassAndSpecificEnergyAbsorption',
    ...
124     '-dpng');
125
126 fig = PlotFactory(3, 10, 0, 0);
127 print(fig, 'AutoPlots/Initialization/WidthAndSpecificEnergyAbsorption',
     ...
128     '-dpng');
129
130 fig = PlotFactory(4, 10, 0, 0);
131 print(fig, ...
132     'AutoPlots/Initialization/NrOfUnitCellsAndSpecificEnergyAbsorption'
        , ...
133     '-dpng');
134
135 fig = PlotFactory(5, 10, 0, 0);
136 print(fig, 'AutoPlots/Initialization/HfAndSpecificEnergyAbsorption',
    ...
137     '-dpng');
138
139 fig = PlotFactory(1, 8, 0, 0);
140 print(fig, 'AutoPlots/Initialization/MassAndBuckling', ...
141     '-dpng');
142
143 fig = PlotFactory(8, 10, 0, 0);
144 print(fig, 'AutoPlots/Initialization/
    BucklingAndSpecificEnergyAbsorption', ...
145     '-dpng');
146
147 fig = PlotFactory(9, 10, 0, 0);
148 print(fig, 'AutoPlots/Initialization/RadiusAndSpecificEnergyAbsorption'
    , ...
149     '-dpng');
150
151 % Plots excluding failing structures
152 plotMode.acceptFailure = 0;
153 fig = PlotFactory(1, 11, 0, 0);
154 print(fig, 'AutoPlots/Initialization/MassAndEnergyAbsorptionNoFailure',
     ...
155     '-dpng');
156
157 fig = PlotFactory(1, 10, 0, 0);
158 print(fig, ...
159     'AutoPlots/Initialization/MassAndSpecificEnergyAbsorptionNoFailure'
        , ...
160     '-dpng');
161
162 fig = PlotFactory(3, 10, 0, 0);
163 print(fig, ...
```

```
164     'AutoPlots/Initialization/WidthAndSpecificEnergyAbsorptionNoFailure
            ', ...
165     '-dpng');
166
167 fig = PlotFactory(4, 10, 0, 0);
168 print(fig, ...
169     'AutoPlots/Initialization/
            NrOfUnitCellsAndSpecificEnergyAbsorptionNoFailure', ...
170     '-dpng');
171
172 fig = PlotFactory(5, 10, 0, 0);
173 print(fig, ...
174     'AutoPlots/Initialization/HfAndSpecificEnergyAbsorptionNoFailure',
            ...
175     '-dpng');
176
177 fig = PlotFactory(1, 8, 0, 0);
178 print(fig, 'AutoPlots/Initialization/MassAndBucklingNoFailure', '-dpng'
        );
179
180 fig = PlotFactory(8, 10, 0, 0);
181 print(fig, ...
182     'AutoPlots/Initialization/
            BucklingAndSpecificEnergyAbsorptionNoFailure', ...
183     '-dpng');
184
185 fig = PlotFactory(9, 10, 0, 0);
186 print(fig, ...
187     'AutoPlots/Initialization/
            RadiusAndSpecificEnergyAbsorptionNoFailure', ...
188     '-dpng');
189
190 % Close all plots
191 close all;
192 end
```

B.3 Optimization algorithm

```
1  function testNr = optimisationAlgorithm(nrOfLevels, heightFactor,
       beamRadius, ...
2      sideWidth, testNr)
3  %-------------------------------------------------------------------------
4  %   PURPOSE
5  %   Optimize input design variables for maximized energy absorption.
6  %
7  %   INPUT
8  %       nrOfLevels = value          Number of unit cells
9  %       heightFactor = value        Height factor
10 %       beamRadius = value          Beam radius [m]
11 %       sideWidth = value           Unit cell width [m]
12 %       testNr = value              Test identification number
13 %
14 %   OUTPUT
```

```
%       testNr = value              Test identification number
%
%-------------------------------------------------------------------------
% Created by Mattias Naarttijarvi
%-------------------------------------------------------------------------

global height testStructure startGuess plotMode

N_tests = (startGuess.N - nrOfLevels.meshSize : nrOfLevels.resolution ...
    : startGuess.N + nrOfLevels.meshSize);

hf_tests = (startGuess.hf - heightFactor.meshSize * heightFactor.resolution ...
    : heightFactor.resolution ...
    : startGuess.hf + heightFactor.meshSize * heightFactor.resolution);

r_tests = (startGuess.r - beamRadius.meshSize * beamRadius.resolution ...
    : beamRadius.resolution ...
    : startGuess.r + beamRadius.meshSize * beamRadius.resolution);

w_tests = (startGuess.w - sideWidth.meshSize * sideWidth.resolution ...
    : sideWidth.resolution ...
    : startGuess.w + sideWidth.meshSize * sideWidth.resolution);

% Keep running until the guess goes through without changing
guessUpdated = 1;
while guessUpdated == 1
    % Start by setting changes to none
    guessUpdated = 0;
    fprintf('\n\nInitializing optimization algorithm\n\n');

    %% Number of unit cells
    fprintf('Varying N \n')

    % Save start test number
    startTestNr = testNr;

    for i = 1 : length(N_tests)
        N = N_tests(i);
        if N > nrOfLevels.max
            % Break if exceeding the maximum
            continue;
        elseif N < nrOfLevels.min
            % Break if below minimum
            continue;
        end

        % Print progress
        fprintf('N = %.0f \n', N);

        % Set each level height
        levelHeights = DescribeLevelHeight(startGuess.hf, N, height);

        % Check if smallest unit cell is large enough
```

```
68          if min(levelHeights) < 6 * startGuess.r
69              fprintf('       Impossible geometry. \n');
70              break;
71          end
72
73          % Do the simulation
74          [Mass, EnergyAbsorption, EnergyAbsorption_F, forceLog, ...
75              displacementLog, sigma, epsilon, buckledLevels] = ...
76              SimulateTest(testStructure, levelHeights, ...
77              startGuess.w, startGuess.r);
78
79          % Evaluate if the tested structure is better then the saved one
80          specificEnergyAbsorption = EnergyAbsorption_F / Mass;
81          if (specificEnergyAbsorption > startGuess.
                specificEnergyAbsorption)
82              % Check if structure withstood the force
83              if (forceLog(end) ~= 0)
84                  fprintf('       Start guess updated by N!\n');
85
86                  % Update the start guess
87                  startGuess.N = N;
88                  startGuess.mass = Mass;
89                  startGuess.sigma = sigma;
90                  startGuess.epsilon = epsilon;
91                  startGuess.force = forceLog;
92                  startGuess.displacement = displacementLog;
93                  startGuess.buckledLevels = buckledLevels;
94                  startGuess.energyAbsorption = EnergyAbsorption_F;
95                  startGuess.specificEnergyAbsorption = ...
96                      specificEnergyAbsorption;
97
98                  disp(startGuess);
99              end
100         end
101
102         % Store data and result
103         StoreData(Mass, 0, startGuess.w, N, ...
104             startGuess.hf, forceLog(end), buckledLevels, startGuess.r,
                    ...
105             EnergyAbsorption_F, testNr, testStructure);
106         testNr = testNr + 1;
107     end
108
109     % Plot and save
110     plotMode.acceptFailure = 1;
111     fig = PlotFactory(4, 10, startTestNr, 0);
112     figName = sprintf('r%.0f w%0.0f hf%1.0f', startGuess.r * 10000, ...
113         startGuess.w * 10000, startGuess.hf*10);
114     print(fig, strcat('AutoPlots/VaryingN/', figName), '-dpng');
115
116     fig = PlotFactory(4, 8, startTestNr, 0);
117     figName = sprintf('r%.0f w%0.0f hf%1.0f', startGuess.r * 10000, ...
118         startGuess.w * 10000, startGuess.hf*10);
119     print(fig, strcat('AutoPlots/VaryingN/', figName, 'Buckling'), '-
            dpng');
120
```

```
121     plotMode.acceptFailure = 0;
122     fig = PlotFactory(4, 8, startTestNr, 0);
123     figName = sprintf('r%.0f w%0.0f hf%1.0f', startGuess.r * 10000, ...
124         startGuess.w * 10000, startGuess.hf*10);
125     print(fig, strcat('AutoPlots/VaryingN/', figName, '
            BucklingNoFailure'), '-dpng');
126
127     fig = PlotFactory(4, 10, startTestNr, 0);
128     figName = sprintf('r%.0f w%0.0f hf%1.0f', startGuess.r * 10000, ...
129         startGuess.w * 10000, startGuess.hf*10);
130     print(fig, strcat('AutoPlots/VaryingN/', figName, 'NoFailure'), '-
            dpng');
131     close all;
132
133     %% Beam radius
134     fprintf('Varying r \n')
135
136     % Save start test number
137     startTestNr = testNr;
138
139     % Set each level height
140     levelHeights = DescribeLevelHeight(startGuess.hf, startGuess.N,
            height);
141
142     for i = 1 : length(r_tests)
143         r = r_tests(i);
144         if r > beamRadius.max
145             % Break if exceeding the maximum
146             continue;
147         elseif r < beamRadius.min
148             % Break if below minimum
149             continue;
150         end
151
152         % Print progress
153         fprintf('r = %.2f mm\n', r*1000);
154
155         % Check if smallest unit cell is large enough
156         if min(levelHeights) < 6 * r
157             fprintf('       Impossible geometry. \n');
158             break;
159         end
160
161         % Do the simulation
162         [Mass, EnergyAbsorption, EnergyAbsorption_F, forceLog, ...
163             displacementLog, sigma, epsilon, buckledLevels] = ...
164             SimulateTest(testStructure, levelHeights, ...
165             startGuess.w, r);
166
167         % Evaluate if the tested structure is better then the saved one
168         specificEnergyAbsorption = EnergyAbsorption_F / Mass;
169         if (specificEnergyAbsorption > startGuess.
                specificEnergyAbsorption)
170             % Check if structure withstood the force
171             if (forceLog(end) ~= 0)
172                 fprintf('       Start guess updated by r!\n');
```

```
173
174                 % Update the start guess
175                 startGuess.r = r;
176                 startGuess.mass = Mass;
177                 startGuess.sigma = sigma;
178                 startGuess.epsilon = epsilon;
179                 startGuess.force = forceLog;
180                 startGuess.buckledLevels = buckledLevels;
181                 startGuess.displacement = displacementLog;
182                 startGuess.energyAbsorption = EnergyAbsorption_F;
183                 startGuess.specificEnergyAbsorption = ...
184                     specificEnergyAbsorption;
185
186                 guessUpdated = 1;
187
188                 disp(startGuess);
189             end
190         end
191
192         % Store data and result
193         StoreData(Mass, 0, startGuess.w, startGuess.N, ...
194             startGuess.hf, forceLog(end), buckledLevels, r, ...
195             EnergyAbsorption_F, testNr, testStructure);
196         testNr = testNr + 1;
197     end
198
199     % Plot and save
200     plotMode.acceptFailure = 1;
201     fig = PlotFactory(9, 10, startTestNr, 0);
202     figName = sprintf('N%.0f w%0.0f hf%1.0f', startGuess.N, ...
203         startGuess.w * 10000, startGuess.hf*10);
204     print(fig, strcat('AutoPlots/VaryingR/', figName), '-dpng');
205
206     fig = PlotFactory(9, 8, startTestNr, 0);
207     figName = sprintf('N%.0f w%0.0f hf%1.0f', startGuess.N, ...
208         startGuess.w * 10000, startGuess.hf*10);
209     print(fig, strcat('AutoPlots/VaryingR/', figName, 'Buckling'), '-
        dpng');
210
211     plotMode.acceptFailure = 0;
212     fig = PlotFactory(9, 8, startTestNr, 0);
213     figName = sprintf('N%.0f w%0.0f hf%1.0f', startGuess.N, ...
214         startGuess.w * 10000, startGuess.hf*10);
215     print(fig, strcat('AutoPlots/VaryingR/', figName, '
        BucklingNoFailure'), '-dpng');
216
217     fig = PlotFactory(9, 10, startTestNr, 0);
218     figName = sprintf('N%.0f w%0.0f hf%1.0f', startGuess.N, ...
219         startGuess.w * 10000, startGuess.hf*10);
220     print(fig, strcat('AutoPlots/VaryingR/', figName, 'NoFailure'), '-
        dpng');
221     close all;
222
223     % Restart the loop
224     if guessUpdated == 1
225         continue;
```

```
226     end
227
228     %% Height factor
229     fprintf('Varying hf \n')
230
231     % Save start test number
232     startTestNr = testNr;
233
234     for i = 1 : length(hf_tests)
235         hf = hf_tests(i);
236         if hf > heightFactor.max
237             % Break if exceeding the maximum
238             continue;
239         elseif hf < heightFactor.min
240             % Break if below minimum
241             continue;
242         end
243
244         % Print progress
245         fprintf('hf = %.1f \n', hf);
246
247         % Set each level height
248         levelHeights = DescribeLevelHeight(hf, startGuess.N, height);
249
250         % Check if smallest unit cell is large enough
251         if min(levelHeights) < 6 * startGuess.r
252             fprintf('       Impossible geometry. \n');
253             break;
254         end
255
256         % Do the simulation
257         [Mass, EnergyAbsorption, EnergyAbsorption_F, forceLog, ...
258             displacementLog, sigma, epsilon, buckledLevels] = ...
259             SimulateTest(testStructure, levelHeights, ...
260             startGuess.w, startGuess.r);
261
262         % Evaluate if the tested structure is better then the saved one
263         specificEnergyAbsorption = EnergyAbsorption_F / Mass;
264         if (specificEnergyAbsorption > startGuess.
                specificEnergyAbsorption)
265             % Check if structure withstood the force
266             if (forceLog(end) ~= 0)
267                 fprintf('       Start guess updated by hf!\n');
268
269                 % Update the start guess
270                 startGuess.hf = hf;
271                 startGuess.mass = Mass;
272                 startGuess.sigma = sigma;
273                 startGuess.epsilon = epsilon;
274                 startGuess.force = forceLog;
275                 startGuess.displacement = displacementLog;
276                 startGuess.buckledLevels = buckledLevels;
277                 startGuess.energyAbsorption = EnergyAbsorption_F;
278                 startGuess.specificEnergyAbsorption =
                        EnergyAbsorption_F / ...
279                     Mass;
```

```
280
281                 guessUpdated = 1;
282
283                 disp(startGuess);
284             end
285         end
286
287         % Store data and result
288         StoreData(Mass, 0, startGuess.w, startGuess.N, ...
289             hf, forceLog(end), buckledLevels, startGuess.r, ...
290             EnergyAbsorption_F, testNr, testStructure);
291         testNr = testNr + 1;
292     end
293
294     % Plot and save
295     plotMode.acceptFailure = 1;
296     fig = PlotFactory(5, 10, startTestNr, 0);
297     figName = sprintf('N%.0f w%0.0f r%1.0f', startGuess.N, ...
298         startGuess.w * 10000, startGuess.r*1000);
299     print(fig, strcat('AutoPlots/VaryingHF/', figName), '-dpng');
300
301     fig = PlotFactory(5, 8, startTestNr, 0);
302     figName = sprintf('N%.0f w%0.0f r%1.0f', startGuess.N, ...
303         startGuess.w * 10000, startGuess.r*1000);
304     print(fig, strcat('AutoPlots/VaryingHF/', figName, 'Buckling'), '-
        dpng');
305
306     plotMode.acceptFailure = 0;
307     fig = PlotFactory(5, 10, startTestNr, 0);
308     figName = sprintf('N%.0f w%0.0f r%1.0f', startGuess.N, ...
309         startGuess.w * 10000, startGuess.r*1000);
310     print(fig, strcat('AutoPlots/VaryingHF/', figName, 'NoFailure'), '-
        dpng')
311
312     fig = PlotFactory(5, 8, startTestNr, 0);
313     figName = sprintf('N%.0f w%0.0f r%1.0f', startGuess.N, ...
314         startGuess.w * 10000, startGuess.r*1000);
315     print(fig, strcat('AutoPlots/VaryingHF/', figName, '
        BucklingNoFailure'), '-dpng');
316     close all;
317
318     % Restart the loop
319     if guessUpdated == 1
320         continue;
321     end
322
323     %% Side width
324     fprintf('Varying w \n')
325
326     % Save start test number
327     startTestNr = testNr;
328
329     % Set each level height
330     levelHeights = DescribeLevelHeight(startGuess.hf, startGuess.N,
        height);
331
```

```
332     % Check if smallest unit cell is large enough
333     if min(levelHeights) < 6 * startGuess.r
334         fprintf('       ERROR! \n');
335         pause(10)
336     end
337 
338     for i = 1 : length(w_tests)
339         w = w_tests(i);
340         if w > sideWidth.max
341             % Break if exceeding the maximum
342             continue;
343         elseif w < sideWidth.min
344             % Break if below minimum
345             continue;
346         end
347 
348         % Print progress
349         fprintf('w = %.1f mm\n', w*1000);
350 
351         % Do the simulation
352         [Mass, EnergyAbsorption, EnergyAbsorption_F, forceLog, ...
353             displacementLog, sigma, epsilon, buckledLevels] = ...
354             SimulateTest(testStructure, levelHeights, ...
355             w, startGuess.r);
356 
357         % Evaluate if the tested structure is better then the saved one
358         specificEnergyAbsorption = EnergyAbsorption_F / Mass;
359         if (specificEnergyAbsorption > startGuess.
                specificEnergyAbsorption)
360             % Check if structure withstood the force
361             if (forceLog(end) ~= 0)
362                 fprintf('       Start guess updated by w!\n');
363 
364                 % Update the start guess
365                 startGuess.w = w;
366                 startGuess.mass = Mass;
367                 startGuess.sigma = sigma;
368                 startGuess.epsilon = epsilon;
369                 startGuess.force = forceLog;
370                 startGuess.displacement = displacementLog;
371                 startGuess.buckledLevels = buckledLevels;
372                 startGuess.energyAbsorption = EnergyAbsorption_F;
373                 startGuess.specificEnergyAbsorption =
                        EnergyAbsorption_F / ...
374                         Mass;
375 
376                 guessUpdated = 1;
377 
378                 disp(startGuess);
379             end
380         end
381 
382         % Store data and result
383         StoreData(Mass, 0, w, startGuess.N, ...
384             startGuess.hf, forceLog(end), buckledLevels, startGuess.r,
                ...
```

```
385             EnergyAbsorption_F, testNr, testStructure);
386         testNr = testNr + 1;
387     end
388 
389     % Plot and save
390     plotMode.acceptFailure = 1;
391     fig = PlotFactory(3, 10, startTestNr, 0);
392     figName = sprintf('N%.0f r%0.0f hf%1.0f', startGuess.N, ...
393         startGuess.r * 10000, startGuess.hf*10);
394     print(fig, strcat('AutoPlots/VaryingW/', figName), '-dpng');
395 
396     fig = PlotFactory(3, 8, startTestNr, 0);
397     figName = sprintf('N%.0f r%0.0f hf%1.0f', startGuess.N, ...
398         startGuess.r * 10000, startGuess.hf*10);
399     print(fig, strcat('AutoPlots/VaryingW/', figName, 'Buckling'), '-
        dpng');
400 
401     plotMode.acceptFailure = 0;
402     fig = PlotFactory(3, 10, startTestNr, 0);
403     figName = sprintf('N%.0f r%0.0f hf%1.0f', startGuess.N, ...
404         startGuess.r * 10000, startGuess.hf*10);
405     print(fig, strcat('AutoPlots/VaryingW/', figName, 'NoFailure'), '-
        dpng');
406 
407     fig = PlotFactory(3, 8, startTestNr, 0);
408     figName = sprintf('N%.0f r%0.0f hf%1.0f', startGuess.N, ...
409         startGuess.r * 10000, startGuess.hf*10);
410     print(fig, strcat('AutoPlots/VaryingW/', figName, '
        BucklingNoFailure'), '-dpng');
411     close all;
412 
413     % Restart the loop
414     if guessUpdated == 1
415         continue;
416     end
417 end
418 end
```

B.4 Simulate compression test

```
1  function [M, EnergyAbsorption, EnergyAbsorption_F, F_list, ...
2      displacement_list, sigma, ...
3      epsilon, buckledLevels] = SimulateTest(chosenStructure, ...
4      levelHeights, sideWidth, R)
5  %-------------------------------------------------------------------------
6  %   PURPOSE
7  %   Simulate a compression test.
8  %
9  %   INPUT
10 %       levelHeights = [h1, h2, ..., hN]        Unit cell height [m]
11 %       sideWidth = value                       Unit cell width [m]
12 %       R = value                               Beam radius [m]
13 %
```

```
%   OUTPUT
%       M = value                   Mass [kg]
%       EnergyAbsorption = value    Energy absorption (s - e) [J]
%       EnergyAbsorption_F = value  Energy absorption (F - delta) [J]
%       F_list = array              Applied force vector
%       displacement_list = array   Displacement vector
%       sigma = array               Stress vector
%       epsilon = array             Strain vector
%       buckledLevels = value       Number of buckled unit cells
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

global E G rho sigma_y height plotMode force

% Structure
    % 1 = tetrahedral
    % 2 = cellulose
    % 3 = pyramid

    % levelHeights = [l1, l2, ..., ln] in m
    % sideWidth = x in m
    % R = x in m, Radius of the beam

% Forces [N]
F = linspace(force.min, force.max, force.steps);

% Force displacement plot
F_list = 0;
displacement_list = 0;

% Anonymous functions
% The cross section area
A = @(r) pi * r^2;

% Mass of a beam
m = @(area, h, rho) area * h * rho;

% Area moment of inertia
I = @(r, m) (m * r^4)/4;

% The moment of inertia, local y-axis
Iy = @(r, h, m) m * (3 * r^3 + h^2) / 12;

% The moment of inertia, local z-axis
Iz = @(r, h, m) m * (3 * r^3 + h^2) / 12;

% Saint-Venant's torsion constant
Kv = @(r) (pi * r^4) / 2;

% Material data
ep = [E G A(R) Iy(R, 0, 0) Iz(R, 0, 0) Kv(R)];

% Sought data
epsilon = 0;
```

```
70 sigma = 0;
71 buckledLevels = 0;
72
73 % Mass of the system
74 [Coord, Edof, Dof, ~, ~, ~] = buildGeometry(...
75             levelHeights, ...
76             sideWidth, ...
77             chosenStructure);
78 [Ex, Ey, Ez] = coordxtr(Edof, Coord, Dof, 2);
79 M = CalcMass(Ex, Ey, Ez, A(R));
80
81 % Step forces
82 f_active = F(1);
83 i = 0;
84 while f_active < force.max
85     % Increase the step
86     i = i + 1;
87     f_active = F(i);
88
89     % Create the geometry
90     if i == 1
91         %fprintf('Building geometry of height %f m \n', sum(
              levelHeights))
92         [Coord, Edof, Dof, unitForce, bc, ~] = buildGeometry(...
93                 levelHeights, ...
94                 sideWidth, ...
95                 chosenStructure);
96     end
97
98     % Active force
99     f = unitForce .* f_active;
100
101    % Solve the stucture
102    [a, ~, Ex, Ey, Ez] = ...
103        SolveStructure(Coord, Edof, Dof, f, bc, ep, R);
104
105    % Store result for force - displacement plot
106    F_list = [F_list, f_active];
107    displacement = height - sum(levelHeights) - a(end - 3);
108    displacement_list = [displacement_list, displacement];
109
110    % Get section forces and displacements
111    Ed = extract(Edof, a);
112    eo = [0, 0, 1];     % Orientation of local z axis
113    eq = [0, 0, 0, 0];  % No distributed load
114    epsilon_test = zeros(length(Ex), 1);
115    sigma_test = zeros(length(Ex), 1);
116
117    % Check if level has buckled
118    buckled = 0;
119
120    for elementNr = 1 : length(Ex)
121        % Beam data
122            ed = Ed(elementNr, :);
123
124            % Position
```

```
125             ex = Ex(elementNr, :);
126             ex_0 = Ex(elementNr, :) + [ed(1), ed(7)];
127             ey = Ey(elementNr, :);
128             ey_0 = Ey(elementNr, :) + [ed(2), ed(8)];
129             ez = Ez(elementNr, :);
130             ez_0 = Ez(elementNr, :) + [ed(3), ed(9)];
131
132             % Undeformed beam length
133             beamLength_0 = sqrt((ex_0(2) - ex_0(1))^2 + ...
134                 (ey_0(2) - ey_0(1))^2 + ...
135                 (ez_0(2) - ez_0(1))^2);
136
137             % Deformed beam length
138             beamLength = sqrt((ex(2) - ex(1))^2 + ...
139                 (ey(2) - ey(1))^2 + ...
140                 (ez(2) - ez(1))^2);
141
142             % Mass
143             beamMass = m(A(R), beamLength_0, rho);
144
145         % Section forces along beam local x-axis
146         % es = [N1 Vy1 Vz1 T1 My1 Mz1;
147         %       N2 Vy2 Vz2 T2 My2 Mz2]
148
149         % Displacements
150         % edi = [u1 v1 w1 fi1;
151         %        u2 v2 w2 fi2]
152
153         % Local x-coordinates for evaluation points
154         % eci = [x1, x2]'
155         ep = [E G A(R) Iy(R, beamLength, beamMass) Iz(R, beamLength,
                ...
156             beamMass) Kv(R)];
157         [es, ~, ~] = beam3s(ex, ey, ez, eo, ep, ed, eq, 2);
158
159         % Difference in deformed and undeformed beam length
160         beamLength_delta = beamLength - beamLength_0;
161         beamStrain = beamLength_delta / beamLength_0;
162         % beamStress = E * beamStrain;
163         beamNormalForce = -es(1, 1);
164         beamStress = beamNormalForce / (pi * R^2);
165
166         % Strain
167         epsilon_test(elementNr) = beamStrain;
168         if beamStress > sigma_y
169             % Elastic perfectly plastic
170             sigma_test(elementNr) = sigma_y;
171         else
172             % Elastic
173             sigma_test(elementNr) = beamStress;
174         end
175
176         % Buckling case 4, fixed in both ends
177         Pk = 4 * pi^2 * E * I(R, beamMass) / (beamLength^2);
178         if beamNormalForce > Pk
179             % Critical load exceeded
```

```
180
181             % Remove top layer
182             nrOfLevels = length(levelHeights);
183             levelHeights(nrOfLevels) = [];
184
185             % Print progress
186             if plotMode.progress == 1
187                 fprintf('   Level nr %1.0f has buckled at F = %1.2f N \
                        n', ...
188                     nrOfLevels, F(i));
189             end
190
191             % Break loop and begin force at force.min again
192             i = 0;
193             buckled = 1;
194             buckledLevels = buckledLevels + 1;
195             break;
196         end
197
198         % Break if all levels have buckeled
199         if isempty(levelHeights)
200             break;
201         elseif buckled == 1
202             break;
203         end
204     end
205
206     % Break if all levels have buckeled
207     if isempty(levelHeights)
208         break;
209     end
210
211     % Store results
212     if sum(epsilon_test) > 0
213         epsilon = [epsilon, sum(epsilon_test)];
214         sigma = [sigma, sum(sigma_test)];
215     end
216 end
217
218 % All layers have buckled
219 if i == 0
220     fprintf('       All unit cells have buckeled!\n');
221     i = 1;
222     F_list(end) = 0;
223 end
224
225 % Calculate energy absorption
226 [EnergyAbsorption, ~] = CalcEnergyAbsorption(sigma, epsilon);
227 [EnergyAbsorption_F, E_list_F] = ...
228     CalcEnergyAbsorption(F_list, displacement_list);
229
230 % Plots
231 if plotMode.results == 1
232     % Sigma epsilon curve
233     figure(4)
234     plot(epsilon, sigma, '-'), hold on;
```

```
235     xlabel('\epsilon')
236     ylabel('\sigma')
237     set(gca, 'fontsize', 18)
238 
239     % Energy curve
240     figure(5)
241     subplot(1, 2, 1);
242     plot(E_list_F, displacement_list, '-'), hold on;
243     subplot(1, 2, 2);
244     plot(E_list_F, F_list, '--'), hold on;
245     ylabel('Energy [J]')
246     set(gca, 'fontsize', 18)
247 
248     % Force vs displacement curve
249     figure(6)
250     plot(displacement_list, F_list), hold on;
251     xlabel('Displacement [m]');
252     ylabel('Force [N]');
253     set(gca, 'fontsize', 18)
254 end
255 end
```

B.5 Solve FEM problem

```
1  function [a, r, Ex, Ey, Ez] = SolveStructure(Coord, Edof, Dof, f, bc,
       ep, R)
2  %-------------------------------------------------------------------------
3  %   PURPOSE
4  %   Solve the FEM problem.
5  %
6  %   INPUT
7  %       Coord = Coordinate matrix
8  %       Edof = Element degree of freedom
9  %       Dof = Degree of freedom
10 %       f = Force vector
11 %       bc = Boundary condition vector
12 %       ep = Material data
13 %       R = Beam radius [m]
14 %
15 %   OUTPUT
16 %       a = Displacement vector
17 %       r = Reaction forces
18 %       Ex = x coordinates
19 %       Ey = y coordinates
20 %       Ez = z coordinates
21 %
22 %-------------------------------------------------------------------------
23 % Created by Alexander Olsson & Mattias Naarttijarvi
24 %-------------------------------------------------------------------------
25 
26 global rho
27 % ep = [E G A Iy Iz Kv]
28 
```

```
% Size of the system
nnodes = length(Coord);              % Number of elements
s = size(Edof);
nel = s(1);
ndeg = 6;                            % Number of degrees of freedom per
    node
ndof = nnodes * ndeg;                % Number of degrees of freedom

% System matrices
K = zeros(ndof, ndof);

% Beam parameters
A = @(r) pi * r^2;                    % The cross section area
m = @(area, h, rho) area * h * rho;   % Mass of a beam

% The moment of inertia
Iy = @(r, h, m) m * ...
    (3 * r^3 + h^2) / 12;
Iz = @(r, h, m) m * (3 * r^3 + h^2) / 12;

% Element properties, topology and coordinates
eo = [0, 0, 1];                  % Orientation of z axis [xz yz zz]
eq = [0, 0, 0, 0];               % Distributed load [qx qy qz qw]

% Extract coordinates
[Ex, Ey, Ez] = coordxtr(Edof, Coord, Dof, 2);

% Assemple element matrices
for i = 1 : nel
    % Length of each beam
    beamLength = sqrt((Ex(i, 2) - Ex(i, 1))^2 + ...
            (Ey(i, 2) - Ey(i, 1))^2 + ...
            (Ez(i, 2) - Ez(i, 1))^2);
    beamMass = m(A(R), beamLength, rho);
    ep(4) = Iy(R, beamLength, beamMass);
    ep(5) = Iz(R, beamLength, beamMass);

    % Assemble
   [Ke, fe] = beam3e(Ex(i, :), Ey(i, :), Ez(i, :), eo, ep, eq);
   [K, f] = assem(Edof(i, :), K, Ke, f, fe);
end

% Solve
[a, r] = solveq(K, f, bc);
end
```

B.6 Minor functions

B.6.1 Mass of the structure

```
function m = CalcMass(Ex, Ey, Ez, A)
%-------------------------------------------------------------------------
%   PURPOSE
```

```
%   Calculate the mass of the entire structure.
%
%   INPUT
%       Ex = x coodinates
%       Ey = y coodinates
%       Ez = z coodinates
%       A = Cross sectional area [m^2]
%
%
%   OUTPUT
%       m = Mass [kg]
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

m = 0;
s = size(Ex);
for i = 1 : s(1)
    % Length of one beam
    lb = sqrt((Ex(i, 2) - Ex(i, 1))^2 + ...
        (Ey(i, 2) - Ey(i, 1))^2 + ...
        (Ez(i, 2) - Ez(i, 1))^2);

    % Mass of a beam
    mb = lb * A;

    % Total mass of all beams
    m = m + mb;
end
end
```

B.6.2 Energy absorption

```
function [E, E_list] = CalcEnergyAbsorption(yValues, xValues)
%-------------------------------------------------------------------------
%   PURPOSE
%   Calculate energy absorption by integrating the input values.
%
%   INPUT
%       yValue = Array
%       xValue = Array
%
%   OUTPUT
%       E = Energy absorption
%       E_list = Energy absorption array in each step
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

% Integrate the stress strain curve
E = 0;
E_list = zeros(1, length(yValues));
```

```
for i = 2 : length(yValues)
    y1 = yValues(i - 1);
    y2 = yValues(i);
    x1 = xValues(i - 1);
    x2 = xValues(i);

    if x2 > x1
        % No buckling
        deltaY = (y2 + y1) / 2;
        deltaX = x2 - x1;

        E = E + deltaY * deltaX;
    else
        % Buckling occured here, take next value
        deltaX = 0;
        deltaY = 0;
    end

    % Store energy for every case
    E_list(i) = deltaY * deltaX;
end
end
```

B.6.3 Unit cell height

```
function levelHeights = DescribeLevelHeight(hf, N, htot)
%-------------------------------------------------------------------------
%   PURPOSE
%   Build the pyramid geometry.
%
%   INPUT
%       hf = Value          Height factor
%       N = Value           Number of unit cells
%       htot = Value        Total height [m]
%
%   OUTPUT
%       levelHeights = [h1, h2, ..., hN]
%
%-------------------------------------------------------------------------
% Created by Mattias Naarttijarvi
%-------------------------------------------------------------------------

hf_sum = 0;
for i = 0 : N - 1
    hf_sum = hf_sum + hf^i;
end

% First level height
h1 = htot / hf_sum;

% Calculate the height variations
levelHeights = zeros(N, 1);
for i = 1 : N
    levelHeights(i) = h1 * hf^(i - 1);
```

```
end
end
```

B.7 Build geometries

B.7.1 Geometry factory

```
function [Coord, Edof, Dof, f, bc, nodes] = buildGeometry(H, W, G)
%-------------------------------------------------------------------------
%   PURPOSE
%   Build the tetrahedron geometry.
%
%   INPUT
%       H = [h1, h2, ..., hn]       Unit cell heights [m]
%       W = value                   Width [m]
%       G = Geometry
%           1 = Tetrahedral
%           2 = Cellulose
%           3 = Pyramid
%           4 = Cube
%
%   OUTPUT
%       Coord = [x1, y1; x2, y2; ...; xn, yn]
%       Edof = Element degree of freedom matrix
%       Dof = Degree of freedom
%       f = Force vector
%       bc = Boundary condition vector
%       nodes = Structure for the geometry
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

% Create the geometry
if G == 1
    % Tetrahedral
    [Coord, Edof, Dof, f, bc, nodes] = ...
        DoTetrahedralGeometry(H, W);
elseif G == 2
    % Cellulose
    [Coord, Edof, Dof, f, bc, nodes] = ...
        DoCellularGeometry(H, W);
elseif G == 3
    % Pyramid
    [Coord, Edof, Dof, f, bc, nodes] = ...
        DoPyramidGeometry(H, W);
elseif G == 4
    % Cube
    [Coord, Edof, Dof, f, bc, nodes] = ...
        DoCubeGeometry(H, W);
end
end
```

B.7.2 Tetrahedron

```
function [Coord, Edof, Dof, f, bc, nodes] = DoTetrahedralGeometry(...
    levelHeights, sideElementWidth)
%-------------------------------------------------------------------------
%   PURPOSE
%   Build the tetrahedron geometry.
%
%   INPUT
%       levelHeights = [h1, h2, ..., hn]        [m]
%       sideElementWidth = value                [m]
%
%   OUTPUT
%       Coord = [x1, y1; x2, y2; ...; xn, yn]
%       Edof = Element degree of freedom matrix
%       Dof = Degree of freedom
%       f = Force vector
%       bc = Boundary condition vector
%       nodes = Structure for the geometry
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

% Initialize
nrOfLevels = length(levelHeights);
accumulatedHeight = 0;
nodeNr = 1;
dofNr = 1;
bcNr = 1;
nedof = 6;
nnodes = (8 * nrOfLevels + 4);
ndof = nnodes * nedof;

% Pre dimensionlize
f = zeros(ndof, 1);
Coord = zeros(nnodes, 3);
Dof = zeros(nnodes, nedof);

% Create each layer
for i = 0 : nrOfLevels - 1
    for iz = 1 : 2
        for iy = 1 : 4
            for ix = 1 : 4
                connections = [];
                noNode = 0;
                if iz == 1
                    % First level, base of tetrahedrals
                    if ix == 1 && iy == 1
                        % 1
                        connections = [...
                            nodeNr + 1, ...
                            nodeNr + 2, ...
                            nodeNr + 4
```

```
53                          ];
54                      elseif ix == 3 && iy == 1
55                          % 2
56                          connections = [...
57                              nodeNr + 1, ...
58                              nodeNr + 2, ...
59                              nodeNr + 3, ...
60                              nodeNr + 4
61                              ];
62                      elseif ix == 2 && iy == 3
63                          % 3
64                          connections = [...
65                              nodeNr + 1, ...
66                              nodeNr + 2, ...
67                              nodeNr + 3, ...
68                              nodeNr + 4, ...
69                              nodeNr + 5
70                          ];
71                      elseif ix == 4 && iy == 3
72                          % 4
73                          connections = [...
74                              nodeNr + 2, ...
75                              nodeNr + 4
76                              ];
77                      else
78                          noNode = 1;
79                      end
80                  elseif iz == 2
81                      % Middle level
82                      if ix == 2 && iy == 2
83                          % 5
84                          connections = [...
85                              nodeNr + 1, ...
86                              nodeNr + 2, ...
87                              nodeNr + 3, ...
88                              nodeNr + 4, ...
89                              nodeNr + 5
90                              ];
91                      elseif ix == 4 && iy == 2
92                          % 6
93                          connections = [...
94                              nodeNr + 2, ...
95                              nodeNr + 4
96                              ];
97                      elseif ix == 1 && iy == 4
98                          % 7
99                          connections = [...
100                             nodeNr + 1, ...
101                             nodeNr + 2, ...
102                             nodeNr + 3, ...
103                             nodeNr + 4
104                             ];
105                     elseif ix == 3 && iy == 4
106                         % 8
107                         connections = [...
108                             nodeNr + 2, ...
```

```
109                             nodeNr + 3, ...
110                             nodeNr + 4
111                         ];
112                     else
113                         noNode = 1;
114                     end
115                 else
116                     % Failsafe
117                     noNode = 1;
118                 end
119 
120                 if noNode == 0
121                     % Boundary conditions
122                     if i == 0 && iz == 1
123                         % Floor
124                         bc(bcNr, :) = [dofNr + 2, 0];
125                         bcNr = bcNr + 1;
126                     end
127 
128                     % Set up node with connections, coordinates and dof
129                     node.x = (ix - 1) / 4 * sideElementWidth;
130                     node.y = (iy - 1) / 4 * sideElementWidth;
131                     node.z = accumulatedHeight + ...
132                         (iz - 1) / 2 * levelHeights(i + 1);
133                     node.dof = dofNr : dofNr + nedof - 1;
134                     node.connections = sort(connections);
135                     node.color = 'o blue';
136 
137                     Coord(nodeNr, :) = [node.x, node.y, node.z];
138                     nodes(nodeNr) = node;
139                     Dof(nodeNr, :) = dofNr : dofNr + nedof - 1;
140 
141                     % Increase indication
142                     nodeNr = nodeNr + 1;
143                     dofNr = dofNr + nedof;
144                 end
145             end
146         end
147     end
148 
149     accumulatedHeight = accumulatedHeight + ...
150                         levelHeights(i + 1);
151 end
152 
153 % Set top level, where the load is applied
154 for iy = 1 : 2
155     for ix = 1 : 4
156         connections = [];
157         noNode = 0;
158         if ix == 1 && iy == 1
159             % 9
160             connections = [...
161                 nodeNr + 1, ...
162                 nodeNr + 2
163                 ];
164         elseif ix == 3 && iy == 1
```

```
165             % 10
166             connections = [...
167                 nodeNr + 1, ...
168                 nodeNr + 2
169                 ];
170         elseif ix == 2 && iy == 2
171             % 11
172             connections = [nodeNr + 1];
173         elseif ix == 4 && iy == 2
174             % 12
175             connections = [];
176         else
177             noNode = 1;
178         end
179 
180         if noNode == 0
181             % Set applied load
182             f(dofNr + 2) = 1;
183 
184             % Set up node with connections, coordinates and dof
185             node.x = (ix - 1) / 4 * sideElementWidth;
186             node.y = (iy - 1) / 2 * sideElementWidth;
187             node.z = sum(levelHeights);
188             node.dof = dofNr : dofNr + nedof - 1;
189             node.connections = sort(connections);
190             node.color = 'o blue';
191 
192             Coord(nodeNr, :) = [node.x, node.y, node.z];
193             nodes(nodeNr) = node;
194             Dof(nodeNr, :) = dofNr : dofNr + nedof - 1;
195 
196             % Increase indication
197             nodeNr = nodeNr + 1;
198             dofNr = dofNr + nedof;
199         end
200     end
201 end
202 
203 % Elements
204 EdofNr = 1;
205 elNr = 1;
206 for i = 1:length(nodes)
207     node = nodes(i);
208     for j = 1:length(node.connections)
209         connectNode = nodes(node.connections(j));
210         if i < node.connections(j)
211             Edof(EdofNr,:) = [EdofNr node.dof connectNode.dof];
212             EdofNr = EdofNr + 1;
213             element.connections = [i, node.connections(j)];
214             element.dof = [Dof(i,:),Dof(node.connections(j),:)];
215             Elements(elNr) = element;
216             elNr = elNr + 1;
217         end
218     end
219 end
220 
```

```
nrOfAppliedLoadNodes = sum(f);
f = -f / nrOfAppliedLoadNodes;

% Dirchlet boundary conditions on all nodes
nDof = length(Coord)*6;
Dirchlet.x = 1 : 6 : nDof;
Dirchlet.y = 2 : 6 : nDof;
Dirchlet.rx = 4 : 6 : nDof;
Dirchlet.ry = 5 : 6 : nDof;
Dirchlet.rz = 6 : 6 : nDof;

% Append bc
for i_bc = 1 : length(Dirchlet.x)
    bc_length = length(bc);
    bc(bc_length + 1, :) = [Dirchlet.x(i_bc), 0];
    bc(bc_length + 2, :) = [Dirchlet.y(i_bc), 0];
    bc(bc_length + 3, :) = [Dirchlet.rx(i_bc), 0];
    bc(bc_length + 4, :) = [Dirchlet.ry(i_bc), 0];
    bc(bc_length + 5, :) = [Dirchlet.rz(i_bc), 0];
end

end
```

B.7.3 Cellulose

```
function [Coord, Edof, Dof, f, bc, nodes] = ...
    DoCellularGeometry(levels, sideElementWidth)
%-------------------------------------------------------------------------
%   PURPOSE
%   Build the cellulose geometry.
%
%   INPUT
%       levelHeights = [h1, h2, ..., hn]        [m]
%       sideElementWidth = value                [m]
%
%   OUTPUT
%       Coord = [x1, y1; x2, y2; ...; xn, yn]
%       Edof = Element degree of freedom matrix
%       Dof = Degree of freedom
%       f = Force vector
%       bc = Boundary condition vector
%       nodes = Structure for the geometry
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson
%-------------------------------------------------------------------------

% Initialize
columns = 2;
rows = 2;
level = 2*length(levels);
nodeNr = 1;
dofNr = 1;
bcNr = 1;
```

```
30 nedof = 6;
31 ndof = (rows - 1)*(columns - 1)*level * 8 * 6;
32
33 f = zeros(ndof,1);
34 accumulatedHeight = 0;
35
36 % Construct nodes
37 for i = 0 : level - 1
38     if mod(i, 2) == 0
39         startX = 0;
40         for l = 0 : 1
41             for j = 0 : columns - 1
42                 for k = 0 : rows - 1
43                     connections = [];
44                     % Boundaries
45                     if j == 0
46                         % Left
47                         if k == 0
48                         % Bottom, j == 0
49                             if l == 0
50                                 if i == 0
51                                     connections = [nodeNr + 1, nodeNr +
                                           ...
52                                         rows, nodeNr + columns * rows];
53                                 else
54                                     connections = [nodeNr + 1, nodeNr +
                                           ...
55                                         rows * columns];
56                                 end
57                             else
58                                 % l == 1;
59                                 if i < level - 1
60                                     connections = [nodeNr + 1, nodeNr +
                                           ...
61                                         columns * rows];
62                                 else
63                                     connections = [nodeNr + 1, nodeNr +
                                           ...
64                                         rows];
65                                 end
66                             end
67                         elseif k == rows - 1
68                             % Top, j == 0
69                             if l == 0
70                                 if i == 0
71                                     connections = [nodeNr + rows,
                                          nodeNr ...
72                                         + columns * rows];
73                                 else
74                                     connections = nodeNr + columns *
                                          rows;
75                                 end
76                             else
77                                 % l == 1;
78                                 if i < level - 1
79                                     connections = nodeNr + columns *
```

```
                                    rows;
80                              else
81                                  connections = nodeNr + rows;
82                              end
83                          end
84
85                      else
86                          % Middle, j == 0
87                          connections = nodeNr + 1;
88                          if l == 0
89                              if i == 0
90                                  connections = [connections, nodeNr
                                        ...
91                                       + rows, nodeNr + columns * rows
                                           ];
92                              else
93                                 connections = [connections, nodeNr +
                                        ...
94                                     columns * rows];
95                              end
96                          else
97                              % l == 1, j == 0
98                              if i < level-1
99                                  connections = [connections, nodeNr
                                        + ...
100                                      columns * rows];
101                             else
102                                 connections = [connections, nodeNr
                                        + ...
103                                      rows];
104                             end
105                         end
106                     end
107                 elseif j == columns - 1
108                     % j == columns - 1
109                     % Right side
110                     if k ~= rows-1
111                          % Top, j == columns - 1
112                         if l == 0
113                             connections = [nodeNr + 1, nodeNr + ...
114                                 columns * rows];
115                         else
116                             % l == 1;
117                             if i < level - 1
118                                 connections = [nodeNr + 1,nodeNr +
                                        ...
119                                      rows * columns - rows, ...
120                                      nodeNr + columns * rows];
121                             else
122                                 connections = nodeNr + 1;
123                             end
124                         end
125
126                     else
127                         if l == 0
128                             connections = nodeNr + rows * columns;
```

```
129                             else
130                                 if i < level - 1
131                                     connections = [nodeNr + columns *
                                            ...
132                                         rows - rows, nodeNr + rows *
                                                ...
133                                         columns];
134                                 else
135                                     % No more connections
136                                 end
137                             end
138                         end
139                     else
140                         % Middle columns
141                          if k ~= rows - 1   % top, j == columns - 1
142                             if l == 0
143                                 if i == 0
144                                     connections = [nodeNr + 1, nodeNr +
                                            ...
145                                         rows, nodeNr + columns * rows];
146                                 else
147                                     connections = [nodeNr + 1, nodeNr +
                                            ...
148                                         columns * rows];
149                                 end
150
151                             else
152                                 % l == 1;
153                                 if i < level - 1
154                                     connections = [nodeNr + 1, nodeNr +
                                            ...
155                                         rows * columns - rows, nodeNr +
                                                ...
156                                         columns * rows];
157                                 else
158                                     connections = [nodeNr + 1, nodeNr +
                                            ...
159                                         rows];
160                                 end
161                             end
162
163                         else
164                           % Top row in middle columns
165                             if l == 0
166                                 if i == 0
167                                     connections = [nodeNr + rows * ...
168                                         columns, nodeNr + rows];
169                                 else
170                                     connections = nodeNr + rows *
                                        columns;
171                                 end
172                             else
173                                 if i < level - 1
174                                     connections = [nodeNr + columns *
                                            ...
175                                         rows - rows, nodeNr + rows *
```

```
176                                         ...
                                        columns];
177                                 else
178                                     connections = nodeNr + rows;
179                                 end
180                             end
181                         end
182                     end
183
184                     % Boundary conditions
185                     if i == 0
186                         bc(bcNr, :) = [dofNr + 2, 0];
187                         bcNr = bcNr + 1;
188                     end
189
190                     % Applied load
191                     if i == level - 1
192                         f(dofNr + 2) = 1;       % Fz
193                     end
194
195                     levelHeight = levels(ceil((i + 1) / 2)) / 2;
196
197                     %Set up the node with coordinates and dof
198                     node.y = sideElementWidth * k;
199                     node.x = j * sideElementWidth + startX;
200                     node.z = accumulatedHeight + l * levelHeight;
201                     node.dof = dofNr : dofNr + nedof - 1;
202                     node.connections = sort(connections);
203                     Dof(nodeNr,:) = dofNr : dofNr + nedof - 1;
204                     node.color = 'o blue';
205                     Coord(nodeNr,:) = [node.x, node.y, node.z];
206                     nodes(nodeNr) = node;
207                     nodeNr = nodeNr + 1;
208                     dofNr = dofNr + nedof;
209
210                 end
211             end
212         end
213     else
214         %------------------------ Next plane
              ----------------------------%
215         startX = sideElementWidth / 2;
216         for l = 0 : 1
217             for j = 0 : columns - 1
218                  for k = 0 : rows - 1
219                     connections = [];
220                     % Boundaries
221                     if j == 0
222                         % Left
223                         if k == 0
224                         % Bottom, j == 0
225                             if l == 0
226                                 connections = [nodeNr + 1, nodeNr + ...
227                                     columns * rows];
228                             else
229                                 if i < level - 1
```

```
230                                 connections = [nodeNr + 1, nodeNr +
                                        ...
231                                     rows * columns, nodeNr + rows +
                                            ...
232                                     rows * columns];
233                             else
234                                 connections = [nodeNr + 1, nodeNr +
                                        ...
235                                     rows];
236                             end
237                         end
238                     elseif k == rows - 1
239                         % Top, j == 0
240                         if l == 0
241                             connections = nodeNr + columns * rows;
242                         else
243                             if i < level - 1
244                                 connections = [nodeNr + rows * ...
245                                     columns, nodeNr + rows + rows *
                                            ...
246                                     columns];
247                             else
248                                 connections = nodeNr + rows;
249                             end
250                         end
251
252                     else
253                         % Middle, j == 0
254                         if l == 0
255                             connections =[nodeNr + 1, nodeNr + ...
256                                 columns * rows];
257                         else
258                             % l == 1, j == 0
259                             if i < level - 1
260                                 connections = [nodeNr + 1, nodeNr +
                                        ...
261                                     columns * rows, nodeNr + ...
262                                     columns * rows + rows];
263                             else
264                                 connections = [nodeNr + 1, ...
265                                     nodeNr + rows];
266                             end
267                         end
268                     end
269                 elseif j == columns - 1
270                     % Right side
271                     if k == 0
272                         % Bottom
273                         if l == 0
274                             connections = [nodeNr + 1, nodeNr + ...
275                                 columns * rows];
276                         else
277                             if i < level - 1
278                                 connections = [nodeNr + 1, nodeNr +
                                        ...
279                                     columns * rows];
```

```
280                             else
281                                 connections = nodeNr + 1;
282                             end
283                         end
284                     elseif k == rows - 1
285                         % Top, j == columns-1
286                         if l == 0
287                             connections = nodeNr + columns * rows;
288                         else
289                             if i < level - 1
290                                 connections = nodeNr + rows *
                                        columns;
291                             else
292                                 % We are at the top and have no
                                        further
293                                 % connections
294                             end
295                         end
296                     else
297                         % Middle rows in j == columns-1
298                         if l == 0
299                             connections = [nodeNr + 1, nodeNr +
                                    rows ...
300                                 * columns];
301                         else
302                             if i < level - 1
303                                 connections = [nodeNr + 1, nodeNr +
                                        ...
304                                     columns * rows];
305                             else
306                                 connections = nodeNr + 1;
307                             end
308                         end
309                     end
310                 else
311                     % Middle columns
312                     if k ~= rows - 1
313                         if l == 0
314                             connections = [nodeNr + 1, nodeNr + ...
315                                 rows * columns];
316                         else
317                             if i < level - 1
318                                 connections = [nodeNr + 1, nodeNr +
                                        ...
319                                     rows * columns, nodeNr + rows *
                                            ...
320                                     columns + rows];
321                             else
322                                 connections = [nodeNr + 1, nodeNr +
                                        ...
323                                     rows];
324                             end
325                         end
326                     else
327                          % We are in top row in middle columns
328                          if l == 0
```

```
329                                 connections = nodeNr + rows * columns;
330                             else
331                                 % l== 1 in middle columns top row
332                                 if i < level - 1
333                                     connections = [nodeNr + columns *
                                            ...
334                                         rows, nodeNr + columns * rows
                                            ...
335                                         + rows];
336                                 else
337                                     connections = nodeNr + rows;
338                                 end
339                             end
340                         end
341                     end
342
343                     % Applied load
344                     if i == level - 1
345                         f(dofNr + 2) = 1;       % Fz
346                     end
347                     levelHeight = levels(ceil((i + 1) / 2)) / 2;
348
349                     % Set up the node with coordinates and dof
350                     node.y = sideElementWidth * k;
351                     node.x = j * sideElementWidth + startX;
352                     node.z = accumulatedHeight + l * levelHeight;
353                     node.dof = dofNr : dofNr + 5;
354                     Dof(nodeNr,:) = dofNr : dofNr + 5;
355                     node.color = 'o red';
356                     Coord(nodeNr,:) = [node.x ,node.y, node.z];
357                     node.connections = sort(connections);
358                     nodes(nodeNr) = node;
359                     nodeNr = nodeNr + 1;
360                     dofNr = dofNr + nedof;
361                 end
362             end
363         end
364     end
365     accumulatedHeight = accumulatedHeight + levels(ceil((i + 1) / 2))
            /2;
366 end
367 EdofNr = 1;
368 for i = 1 : length(nodes)
369     node = nodes(i);
370     for j = 1 : length(node.connections)
371         connectNode = nodes(node.connections(j));
372         if i < node.connections(j)
373             Edof(EdofNr, :) = [EdofNr node.dof connectNode.dof];
374             EdofNr = EdofNr + 1;
375             element.connections = [i, node.connections(j)];
376             element.dof = [Dof(i, :), Dof(node.connections(j), :)];
377         end
378     end
379 end
380
381 % Force
```

```
382 nrOfAppliedLoadNodes = sum(f);
383 f = -f / nrOfAppliedLoadNodes;
384
385 % Dirchlet bouncary condition
386 nDof = length(Coord)*6;
387 Dirchlet.x = 1 : 6 : nDof;
388 Dirchlet.y = 2 : 6 : nDof;
389 Dirchlet.rx = 4 : 6 : nDof;
390 Dirchlet.ry = 5 : 6 : nDof;
391 Dirchlet.rz = 6 : 6 : nDof;
392
393 % Append bc
394 for i_bc = 1 : length(Dirchlet.x)
395     bc_length = length(bc);
396     bc(bc_length + 1, :) = [Dirchlet.x(i_bc), 0];
397     bc(bc_length + 2, :) = [Dirchlet.y(i_bc), 0];
398     bc(bc_length + 3, :) = [Dirchlet.rx(i_bc), 0];
399     bc(bc_length + 4, :) = [Dirchlet.ry(i_bc), 0];
400     bc(bc_length + 5, :) = [Dirchlet.rz(i_bc), 0];
401 end
```

B.7.4 Pyramid

```
1  function [Coord, Edof, Dof, f, bc, nodes] = DoPyramidGeometry(...
2      levelHeights, sideElementWidth)
3  %-------------------------------------------------------------------------
4  %   PURPOSE
5  %   Build the pyramid geometry.
6  %
7  %   INPUT
8  %       levelHeights = [h1, h2, ..., hn]        [m]
9  %       sideElementWidth = value                [m]
10 %
11 %   OUTPUT
12 %       Coord = [x1, y1; x2, y2; ...; xn, yn]
13 %       Edof = Element degree of freedom matrix
14 %       Dof = Degree of freedom
15 %       f = Force vector
16 %       bc = Boundary condition vector
17 %       nodes = Structure for the geometry
18 %
19 %-------------------------------------------------------------------------
20 % Created by Mattias Naarttijarvi
21 %-------------------------------------------------------------------------
22
23
24 % Initialize
25 nrOfLevels = length(levelHeights);
26 accumulatedHeight = 0;
27 nodeNr = 1;
28 dofNr = 1;
29 bcNr = 1;
30 nedof = 6;
31 nnodes = (9 * nrOfLevels + 4);
```

```
ndof = nnodes * nedof;

% Pre dimensionlize
f = zeros(ndof, 1);
Coord = zeros(nnodes, 3);
Dof = zeros(nnodes, nedof);

% Create each layer
for i = 0 : nrOfLevels - 1
    for iz = 1 : 2
        for iy = 1 : 3
            for ix = 1 : 3
                connections = [];
                noNode = 0;
                if iz == 1
                    % Square level, base of pyramid
                    if iy == 1 && ix == 1
                        % First corner
                        connections = [ ...
                            nodeNr + 1, ... % Next corner
                            nodeNr + 2, ... % Previous corner
                            nodeNr + 6, ... % Center of next level
                            nodeNr + 4];    % Diagonal beam
                    elseif iy == 1 && ix == 3
                        % Second corner
                        connections = [ ...
                            nodeNr + 2, ... % Next corner
                            nodeNr + 5, ... % Center of next level
                            nodeNr + 6];    % Diagonal beam
                    elseif iy == 3 && ix == 1
                        % Third corner
                        connections = [ ...
                            nodeNr + 1, ... % Next corner
                            nodeNr + 4, ... % Center of next level
                            nodeNr + 3];    % Diagonal beam
                    elseif iy == 3 && ix == 3
                        % Forth corner
                        connections = [ ...
                            nodeNr + 3, ... % Center of next level
                            nodeNr + 5];    % Diagonal beam
                    else
                        noNode = 1;
                    end
                elseif iz == 2
                    % Middle level, top of pyramid
                    if iy == 1 && ix == 2
                        connections = [nodeNr + 6];
                    elseif iy == 2 && ix == 1
                        connections = [nodeNr + 4];
                    elseif iy == 2 && ix == 2
                        % Center node
                        connections = [...
                            nodeNr + 3, ...
                            nodeNr + 4, ...
                            nodeNr + 5, ...
                            nodeNr + 6];
```

```
88                      elseif iy == 2 && ix == 3
89                          connections = [nodeNr + 5];
90                      elseif iy == 3 && ix == 2
91                          connections = [nodeNr + 3];
92                      else
93                          noNode = 1;
94                      end
95                  else
96                      noNode = 1;
97                  end
98
99                  if noNode == 0
100                     % Boundary conditions
101                     if i == 0 && iz == 1
102                         % Floor
103                         bc(bcNr, :) = [dofNr + 2, 0];
104                         bcNr = bcNr + 1;
105                     end
106                     if ix ~= 2 && iy ~= 2
107                         % Dirchlet boundary conditions at edge nodes
108                         bc(bcNr, :) = [dofNr, 0];
109                         bc(bcNr + 1, :) = [dofNr + 1, 0];
110                         bc(bcNr + 2, :) = [dofNr + 3, 0];
111                         bc(bcNr + 3, :) = [dofNr + 4, 0];
112                         bc(bcNr + 4, :) = [dofNr + 5, 0];
113                         bcNr = bcNr + 5;
114                     end
115
116                     % Set up node with connections, coordinates and dof
117                     node.x = (ix - 1) / 2 * sideElementWidth;
118                     node.y = (iy - 1) / 2 * sideElementWidth;
119                     node.z = accumulatedHeight + ...
120                         (iz - 1) / 2 * levelHeights(i + 1);
121                     node.dof = dofNr : dofNr + nedof - 1;
122                     node.connections = sort(connections);
123                     node.color = 'o blue';
124
125                     Coord(nodeNr, :) = [node.x, node.y, node.z];
126                     nodes(nodeNr) = node;
127                     Dof(nodeNr, :) = dofNr : dofNr + nedof - 1;
128
129                     % Increase indication
130                     nodeNr = nodeNr + 1;
131                     dofNr = dofNr + nedof;
132                 end
133             end
134         end
135     end
136
137     accumulatedHeight = accumulatedHeight + ...
138                         levelHeights(i + 1);
139 end
140
141 % Set top level, where the load is applied
142 for iy = 1 : 2
143      for ix = 1 : 2
```

```
144         connections = [];
145         if ix == 1 && iy == 1
146             % First corner
147             connections = [...
148                 nodeNr + 1, ...
149                 nodeNr + 2];
150         elseif ix == 1 && iy == 2
151             connections = [nodeNr + 1];
152         elseif ix == 2 && iy == 1
153             connections = [nodeNr + 2];
154         end
155 
156         % Dirchlet boundary conditions at edge nodes
157         bc(bcNr, :) = [dofNr, 0];
158         bc(bcNr + 1, :) = [dofNr + 1, 0];
159         bc(bcNr + 2, :) = [dofNr + 3, 0];
160         bc(bcNr + 3, :) = [dofNr + 4, 0];
161         bc(bcNr + 4, :) = [dofNr + 5, 0];
162         bcNr = bcNr + 5;
163 
164 
165         % Set applied load
166         f(dofNr + 2) = 1;
167 
168         % Set up node with connections, coordinates and dof
169         node.x = (ix - 1) * sideElementWidth;
170         node.y = (iy - 1) * sideElementWidth;
171         node.z = sum(levelHeights);
172         node.dof = dofNr : dofNr + nedof - 1;
173         node.connections = sort(connections);
174         node.color = 'o blue';
175 
176         Coord(nodeNr, :) = [node.x, node.y, node.z];
177         nodes(nodeNr) = node;
178         Dof(nodeNr, :) = dofNr : dofNr + nedof - 1;
179 
180         % Increase indication
181         nodeNr = nodeNr + 1;
182         dofNr = dofNr + nedof;
183     end
184 end
185 
186 
187 % Elements
188 EdofNr = 1;
189 elNr = 1;
190 for i = 1:length(nodes)
191     node = nodes(i);
192     for j = 1:length(node.connections)
193         connectNode = nodes(node.connections(j));
194         if i < node.connections(j)
195             Edof(EdofNr,:) = [EdofNr node.dof connectNode.dof];
196             Edof2(EdofNr,:) = [i,node.connections(j)];
197             EdofNr = EdofNr + 1;
198             element.connections = [i, node.connections(j)];
199             element.dof = [Dof(i,:),Dof(node.connections(j),:)];
```

```
200             Elements(elNr) = element;
201             elNr = elNr + 1;
202         end
203     end
204 end
205 
206 % Force
207 nrOfAppliedLoadNodes = sum(f);
208 f = -f / nrOfAppliedLoadNodes;
209 
210 end
```

B.7.5 Cube

```
1  function [Coord, Edof, Dof, f, bc, nodes] = ...
2      DoCubeGeometry(levelHeights, sideElementWidth)
3  %--------------------------------------------------------------------------
4  %   PURPOSE
5  %   Build the cube geometry.
6  %
7  %   INPUT
8  %       levelHeights = [h1, h2, ..., hn]        [m]
9  %       sideElementWidth = value                [m]
10 %
11 %   OUTPUT
12 %       Coord = [x1, y1; x2, y2; ...; xn, yn]
13 %       Edof = Element degree of freedom matrix
14 %       Dof = Degree of freedom
15 %       f = Force vector
16 %       bc = Boundary condition vector
17 %       nodes = Structure for the geometry
18 %
19 %--------------------------------------------------------------------------
20 % Created by Alexander Olsson
21 %--------------------------------------------------------------------------
22 
23 % Initialize
24 columns = 2;
25 rows = 2;
26 level = length(levelHeights);
27 levelH = [0 levelHeights'];
28 nodeNr = 1;
29 dofNr = 1;
30 bcNr = 1;
31 nedof = 6;
32 nnodes = rows * columns * (level + 1);
33 ndof = nnodes * 6;
34 accumulatedHeight = 0;
35 
36 % Pre dimensionlize
37 f = zeros(ndof, 1);
38 Coord = zeros(nnodes, 3);
39 Dof = zeros(nnodes, nedof);
40 
```

```
41 for i = 1 : level + 1
42     for j = 0 : columns - 1
43         for k = 0 : rows - 1
44             connections = [];
45                 % Left
46                 if j == 0
47                     % Bottom
48                     if k == 0
49                         if i ~= level + 1
50                             connections = [nodeNr + 1, nodeNr + rows,
                                   ...
51                                 nodeNr + columns * rows];
52                         else
53                              connections = [nodeNr + 1, nodeNr + rows];
54                         end
55                     % Top
56                     elseif k == rows - 1
57                          if i ~= level + 1
58                             connections = [nodeNr + rows, nodeNr + ...
59                                 columns * rows];
60                          else
61                             connections = nodeNr + rows;
62                          end
63                     end
64                 % Right
65                 elseif j == columns - 1
66                     % Bottom
67                     if k == 0
68                         if i ~= level + 1
69                             connections = [nodeNr + 1, nodeNr + ...
70                                 columns * rows];
71                         else
72                             connections = nodeNr + 1;
73                         end
74                     % Top
75                     elseif k == rows - 1
76                         if i ~= level + 1
77                             connections = nodeNr + columns * rows;
78                         end
79                     end
80                 end
81
82                 % Applied force
83                 if i == level + 1
84                     f(dofNr + 2, :) = 1;
85                 end
86
87                 % Set up the node with coordinates and dof
88                 node.y = sideElementWidth * k;
89                 node.x = j * sideElementWidth;
90                 node.z = accumulatedHeight + levelH(i);
91                 node.dof = dofNr : dofNr + nedof - 1;
92                 node.connections = sort(connections);
93                 Dof(nodeNr, :) = dofNr : dofNr + nedof - 1;
94                 Coord(nodeNr, :) = [node.x, node.y, node.z];
95                 nodes(nodeNr) = node;
```

```
                nodeNr = nodeNr + 1;

                % Boundary conditions
                if i == 1
                    bc(bcNr, :) = [dofNr 0];
                    bc(bcNr + 1, :) = [dofNr + 1 0];
                    bc(bcNr + 2, :) = [dofNr + 2 0];
                    bcNr = bcNr + 3;
                else
                    bc(bcNr, :) = [dofNr 0];
                    bc(bcNr + 1, :) = [dofNr + 1 0];
                    bcNr = bcNr + 2;
                end
                dofNr = dofNr + nedof;
            end
        end
        accumulatedHeight = accumulatedHeight + levelH(i);
    end

    % Elements
    EdofNr = 1;
    for i = 1:length(nodes)
        node = nodes(i);
        for j = 1:length(node.connections)
            connectNode = nodes(node.connections(j));
            if i < node.connections(j)
                Edof(EdofNr,:) = [EdofNr node.dof connectNode.dof];
                EdofNr = EdofNr + 1;
                element.connections = [i, node.connections(j)];
                element.dof = [Dof(i,:),Dof(node.connections(j),:)];
            end
        end
    end

    % Force
    nrOfAppliedLoadNodes = sum(f);
    f = -f / nrOfAppliedLoadNodes;
    end
```

B.8 Plot and store data

B.8.1 Generate plot

```
function fig = PlotFactory(x, y, startIndex, endIndex)
%-------------------------------------------------------------------------
%   PURPOSE
%   Plots a 2D graph of desired test data for global test structure.
%
%   Value on axis
%   1 = Mass, 2 = Energy_s, 3 = Width, 4 = Number of unit cells,
%   5 = Height factor,  6 = Max load
%   7 = Geometry, 8 = Buckled unit cells, 9 = Radius,
%   10 = Specific energy absorption, 11 = Energy absorption
```

```
11 %
12 %   INPUT
13 %       x = value on x axis
14 %       y = value on y axis
15 %       startIndex = Test number to start with, 0 if first
16 %       endIndex = Test number to start with, 0 is last
17 %
18 %   OUTPUT
19 %       fig = The figure
20 %
21 %--------------------------------------------------------------------------
22 % Created by Mattias Naarttijarvi
23 %--------------------------------------------------------------------------
24
25 global mainData testStructure plotMode;
26 acceptFailure = plotMode.acceptFailure;
27 figureNr = plotMode.figureNr;
28
29 if (startIndex == 0)
30     startIndex = 1;
31 end
32
33 fig = figure(figureNr);
34 for i_test = 1:length(testStructure)
35     % Tetrahedral
36     if (testStructure(i_test) == 1)
37         tetrahedronPassed.x = [];
38         tetrahedronPassed.y = [];
39         tetrahedronFailed.x = [];
40         tetrahedronFailed.y = [];
41
42         if (endIndex == 0)
43             endIndex = length(mainData.tetrahedral(:, x));
44         end
45
46         for i = startIndex :  endIndex
47             if (mainData.tetrahedral(i, 6) == 0)
48                 % Failed tests
49                 if (acceptFailure == 1)
50                     tetrahedronFailed.x = [tetrahedronFailed.x, ...
51                         mainData.tetrahedral(i, x)];
52                     tetrahedronFailed.y = [tetrahedronFailed.y, ...
53                         mainData.tetrahedral(i, y)];
54                 end
55             else
56                 % Passed tests
57                 tetrahedronPassed.x = [tetrahedronPassed.x, ...
58                         mainData.tetrahedral(i, x)];
59                 tetrahedronPassed.y = [tetrahedronPassed.y, ...
60                         mainData.tetrahedral(i, y)];
61             end
62         end
63
64         % Create the plots
65         if ~isempty(tetrahedronPassed.x)
66             plot(tetrahedronPassed.x, tetrahedronPassed.y, 'bo');
```

```
67              hold on;
68          end
69          if ~isempty(tetrahedronFailed.x)
70              plot(tetrahedronFailed.x, tetrahedronFailed.y, 'b*');
71              hold on;
72          end
73      end
74
75      % Cellulose
76      if (testStructure(i_test) == 2)
77          cellulosePassed.x = [];
78          cellulosePassed.y = [];
79          celluloseFailed.x = [];
80          celluloseFailed.y = [];
81
82          if (endIndex == 0)
83              endIndex = length(mainData.cellulose(:, x));
84          end
85
86          for i = startIndex :  endIndex
87              if (mainData.cellulose(i, 6) == 0)
88                  % Failed tests
89                  if (acceptFailure == 1)
90                      celluloseFailed.x = [celluloseFailed.x, ...
91                          mainData.cellulose(i, x)];
92                      celluloseFailed.y = [celluloseFailed.y, ...
93                          mainData.cellulose(i, y)];
94                  end
95              else
96                  % Passed tests
97                  cellulosePassed.x = [cellulosePassed.x, ...
98                          mainData.cellulose(i, x)];
99                  cellulosePassed.y = [cellulosePassed.y, ...
100                         mainData.cellulose(i, y)];
101             end
102         end
103
104         % Create the plots
105         if ~isempty(cellulosePassed.x)
106             plot(cellulosePassed.x, cellulosePassed.y, 'bo');
107             hold on;
108         end
109         if ~isempty(celluloseFailed.x)
110             plot(celluloseFailed.x, celluloseFailed.y, 'b*');
111             hold on;
112         end
113     end
114
115     % Pyramid
116     if (testStructure(i_test) == 3)
117         pyramidPassed.x = [];
118         pyramidPassed.y = [];
119         pyramidFailed.x = [];
120         pyramidFailed.y = [];
121
122         if (endIndex == 0)
```

```
123             endIndex = length(mainData.pyramid(:, x));
124         end
125 
126         for i = startIndex :  endIndex
127             if (mainData.pyramid(i, 6) == 0)
128                 % Failed tests
129                 if (acceptFailure == 1)
130                     pyramidFailed.x = [pyramidFailed.x, ...
131                         mainData.pyramid(i, x)];
132                     pyramidFailed.y = [pyramidFailed.y, ...
133                         mainData.pyramid(i, y)];
134                 end
135             else
136                 % Passed tests
137                 pyramidPassed.x = [pyramidPassed.x, ...
138                         mainData.pyramid(i, x)];
139                 pyramidPassed.y = [pyramidPassed.y, ...
140                         mainData.pyramid(i, y)];
141             end
142         end
143 
144         % Create the plots
145         if ~isempty(pyramidPassed.x)
146             plot(pyramidPassed.x, pyramidPassed.y, 'bo');
147             hold on;
148         end
149         if ~isempty(pyramidFailed.x)
150             plot(pyramidFailed.x, pyramidFailed.y, 'b*');
151             hold on;
152         end
153     end
154 
155     % Cube
156     if (testStructure(i_test) == 4)
157         cubePassed.x = [];
158         cubePassed.y = [];
159         cubeFailed.x = [];
160         cubeFailed.y = [];
161 
162         if (endIndex == 0)
163             endIndex = length(mainData.cube(:, x));
164         end
165 
166         for i = startIndex :  endIndex
167             if (mainData.cube(i, 6) == 0)
168                 % Failed tests
169                 if (acceptFailure == 1)
170                     cubeFailed.x = [cubeFailed.x, ...
171                         mainData.cube(i, x)];
172                     cubeFailed.y = [cubeFailed.y, ...
173                         mainData.cube(i, y)];
174                 end
175             else
176                 % Passed tests
177                 cubePassed.x = [cubePassed.x, ...
178                         mainData.cube(i, x)];
```

```
179                 cubePassed.y = [cubePassed.y, ...
180                         mainData.cube(i, y)];
181             end
182         end
183 
184         % Create the plots
185         if ~isempty(cubePassed.x)
186             plot(cubePassed.x, cubePassed.y, 'bo');
187             hold on;
188         end
189         if ~isempty(cubeFailed.x)
190             plot(cubeFailed.x, cubeFailed.y, 'b*');
191             hold on;
192         end
193     end
194 end
195 
196 % Set axis labels
197 switch x
198     case 1
199         xlabel('Mass [kg]');
200     case 2
201         xlabel('Energy absorption from sigma - epsilon [J/kg]');
202     case 3
203         xlabel('Width [m]');
204     case 4
205         xlabel('Number of unit cells');
206     case 5
207         xlabel('Height factor');
208     case 6
209         xlabel('Max load [F]');
210     case 7
211         xlabel('Geometry');
212     case 8
213         xlabel('Buckled unit cells');
214     case 9
215         xlabel('Beam radius [m]')
216     case 10
217         xlabel('Specific energy absorption [J/kg]')
218     case 11
219         xlabel('Energy absorption [J]')
220 end
221 switch y
222     case 1
223         ylabel('Mass [kg]');
224     case 2
225         ylabel('Energy absorption from sigma - epsilon [J/kg]');
226     case 3
227         ylabel('Width [m]');
228     case 4
229         ylabel('Number of unit cells');
230     case 5
231         ylabel('Height factor');
232     case 6
233         ylabel('Max load [F]');
234     case 7
```

```
235         ylabel('Geometry');
236     case 8
237         ylabel('Buckled unit cells');
238     case 9
239         ylabel('Beam radius [m]')
240     case 10
241         ylabel('Specific energy absorption [J/kg]')
242     case 11
243         ylabel('Energy absorption [J]')
244 end
245
246 % Legend
247 figure(figureNr)
248 grid on;
249 if (max(testStructure == 1) == 1 && length(testStructure) == 1)
250     % Tetrahedron
251     if ~isempty(tetrahedronPassed.x) && ~isempty(tetrahedronFailed.x)
252         % Both passed and failed
253         legend('Tetrahedron', 'Tetrahedron failed', 'location', '
                northoutside')
254     elseif ~isempty(tetrahedronPassed.x)
255         % Only passed
256         legend('Tetrahedron', 'location', 'northoutside')
257     elseif ~isempty(tetrahedronFailed.x)
258         % Only failed
259         legend('Tetrahedron failed', 'location', 'northoutside')
260     end
261
262 elseif (max(testStructure == 2) == 1 && length(testStructure) == 1)
263     % Cellulose
264     if ~isempty(cellulosePassed.x) && ~isempty(celluloseFailed.x)
265         % Both passed and failed
266         legend('Cellulose', 'Cellulose failed', 'location', '
                northoutside')
267     elseif ~isempty(cellulosePassed.x)
268         % Only passed
269         legend('Cellulose', 'location', 'northoutside')
270     elseif ~isempty(celluloseFailed.x)
271         % Only failed
272         legend('Cellulose failed', 'location', 'northoutside')
273     end
274
275 elseif (max(testStructure == 3) == 1 && length(testStructure) == 1)
276     % Pyramid
277     if ~isempty(pyramidPassed.x) && ~isempty(pyramidFailed.x)
278         % Both passed and failed
279         legend('Pyramid', 'Pyramid failed', 'location', 'northoutside')
280     elseif ~isempty(pyramidPassed.x)
281         % Only passed
282         legend('Pyramid', 'location', 'northoutside')
283     elseif ~isempty(pyramidFailed.x)
284         % Only failed
285         legend('Pyramid failed', 'location', 'northoutside')
286     end
287
288 elseif (max(testStructure == 4) == 1 && length(testStructure) == 1)
```

```
289     % Cube
290     if ~isempty(cubePassed.x) && ~isempty(cubeFailed.x)
291         % Both passed and failed
292         legend('Cube', 'Cube failed', 'location', 'northoutside')
293     elseif ~isempty(cubePassed.x)
294         % Only passed
295         legend('Cube', 'location', 'northoutside')
296     elseif ~isempty(celluloseFailed.x)
297         % Only failed
298         legend('Cube failed', 'location', 'northoutside')
299     end
300 end
301 set(gca, 'fontsize', 18)
302 
303 plotMode.figureNr = figureNr + 1;
304 end
```

B.8.2 Plot geometry

```
1  function fig = PlotGeometry(chosenStructure, w, N, hf)
2  %-------------------------------------------------------------------------
3  %   PURPOSE
4  %   Plot a 3D figure of the structure
5  %
6  %   INPUT
7  %       chosenStructure = Value 1 to 4
8  %       w = value           Unit cell width [m]
9  %       N = value           Number of unit cells
10 %       hf = value          Height factor
11 %
12 %
13 %   OUTPUT
14 %       fig = The figure
15 %
16 %-------------------------------------------------------------------------
17 % Created by Alexander Olsson & Mattias Naarttijarvi
18 %-------------------------------------------------------------------------
19 
20 global plotMode height
21 
22 % Set each level height
23 levelHeights = DescribeLevelHeight(hf, N, height);
24 
25 % Mass of the system
26 [Coord, Edof, Dof, ~, ~, ~] = buildGeometry(...
27             levelHeights, ...
28             w, ...
29             chosenStructure);
30 [Ex, Ey, Ez] = coordxtr(Edof, Coord, Dof, 2);
31 
32 % Structure and deformed mesh of best solution
33 plotMode.figureNr = plotMode.figureNr + 1;
34 fig = figure(plotMode.figureNr);
35 eldraw3(Ex, Ey, Ez, [1 1 0]), hold on;
```

```
grid on;
set(gca, 'fontsize', 18)
end
```

```
function PlotStructure(Ex, Ey, Ez, Edof, a)
%-------------------------------------------------------------------------
%   PURPOSE
%   Plot a 3D figure of the structure with displacement
%
%   INPUT
%       Ex = x coordinates
%       Ey = y coordinates
%       Ez = z coordinates
%       Edof = Element degree of freedom
%       a = Displacement vector
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson & Mattias Naarttijarvi
%-------------------------------------------------------------------------

% Draw the structure
eldraw3(Ex, Ey, Ez);

% Extract element displacement and display deformed mesh
Ed = extract(Edof, a);
eldisp3(Ex, Ey, Ez, Ed, [2 4 1]);
end
```

B.8.3 Store data

```
function StoreData(m, e, w, N, hf, f, b, r, e_F, testNr, testStructure)
%-------------------------------------------------------------------------
%   PURPOSE
%   Store data into global mainData structure.
%
%   INPUT
%       m = value                       Mass [kg]
%       e = value                       Energy absorption sigma - eps. [J
    ]
%       w = value                       Unit cell width [m]
%       N = value                       Number of unit cells
%       hf = value                      Height factor
%       f = value                       Max load [N]
%       b = value                       Number of buckled unit cells
%       r = value                       Beam radius [m]
%       e_F = value                     Energy absorption F - delta [J]
%       testNr = value                  Test identification number
%       testStructure = value           1 - 4
%
%-------------------------------------------------------------------------
% Created by Alexander Olsson
%-------------------------------------------------------------------------
global mainData
```

```
23
24 % Store data and results
25 if testStructure == 1
26     % 1 = tetrahedral
27     mainData.tetrahedral(testNr,1) = m;
28     mainData.tetrahedral(testNr,2) = e;
29     mainData.tetrahedral(testNr,3) = w;
30     mainData.tetrahedral(testNr,4) = N;
31     mainData.tetrahedral(testNr,5) = hf;
32     mainData.tetrahedral(testNr,6) = f;
33     mainData.tetrahedral(testNr,7) = 1;
34     mainData.tetrahedral(testNr,8) = b;
35     mainData.tetrahedral(testNr,9) = r;
36     mainData.tetrahedral(testNr,10) = e_F / m;
37     mainData.tetrahedral(testNr,11) = e_F;
38
39 elseif testStructure == 2
40     % 2 = cellulose
41     mainData.cellulose(testNr,1) = m;
42     mainData.cellulose(testNr,2) = e;
43     mainData.cellulose(testNr,3) = w;
44     mainData.cellulose(testNr,4) = N;
45     mainData.cellulose(testNr,5) = hf;
46     mainData.cellulose(testNr,6) = f;
47     mainData.cellulose(testNr,7) = 2;
48     mainData.cellulose(testNr,8) = b;
49     mainData.cellulose(testNr,9) = r;
50     mainData.cellulose(testNr,10) = e_F / m;
51     mainData.cellulose(testNr,11) = e_F;
52
53 elseif testStructure == 3
54     % 3 = pyramid
55     mainData.pyramid(testNr,1) = m;
56     mainData.pyramid(testNr,2) = e;
57     mainData.pyramid(testNr,3) = w;
58     mainData.pyramid(testNr,4) = N;
59     mainData.pyramid(testNr,5) = hf;
60     mainData.pyramid(testNr,6) = f;
61     mainData.pyramid(testNr,7) = 3;
62     mainData.pyramid(testNr,8) = b;
63     mainData.pyramid(testNr,9) = r;
64     mainData.pyramid(testNr,10) = e_F / m;
65     mainData.pyramid(testNr,11) = e_F;
66
67 elseif testStructure == 4
68     % 4 = cube
69     mainData.cube(testNr,1) = m;
70     mainData.cube(testNr,2) = e;
71     mainData.cube(testNr,3) = w;
72     mainData.cube(testNr,4) = N;
73     mainData.cube(testNr,5) = hf;
74     mainData.cube(testNr,6) = f;
75     mainData.cube(testNr,7) = 3;
76     mainData.cube(testNr,8) = b;
77     mainData.cube(testNr,9) = r;
78     mainData.cube(testNr,10) = e_F / m;
```

```
79      mainData.cube(testNr,11) = e_F;
80  end
81  end
```