

Computational Evaluation and Optimization of Mechanical and Form Properties of 3D- Printed Bone Tissue Engineering Scaffolds

Author: Max Engensperger

Supervised by: Dr. Paul Egan

Master's Thesis

FS 2016

ABSTRACT

This thesis is about computational mechanical simulation and optimization of scaffold designs for bone tissue engineering. The scaffolds consist of repeating unit cells which are made of struts creating a lattice structure with discrete elements. Several different unit cells are mechanically evaluated and compared via the finite element method. The simulation makes use of Euler-Bernoulli beam elements which represent the scaffold as a wire frame.

As design variables the strut thickness of each strut of a unit cell and the overall cell size of the unit cells are used. As optimization goal, the biomimetic approach is set, to match the compliance of the scaffold with human cortical bone.

It was determined that the wire frame approximation allows for fast mechanical evaluation of scaffolds, however leads to inaccuracy because of beam elements. The resulting scaffolds have matching mechanical properties with bone, but would have to be tested empirically in mechanical loading experiments to confirm this statement.

Design automation is achieved through the programming language Python, using the Python distribution Anaconda, and the evaluation is executed with the software Abaqus from Dassault Systèmes.

The code that was written for this thesis can be retrieved at
<https://github.com/Engensmax/Truss-Builder>.

ACKNOWLEDGEMENTS

I would like to thank Dr. Paul Egan for supporting this thesis from beginning to end.

My thanks to Cynthia Tan for handling the 3d-printing of some examples for me.

My thanks to Prof. Dr. Kristina Shea for the opportunity to write my master's thesis at the Engineering Design and Computing Lab (EDAC).

A thank you to the community of stack overflow for providing answers to my questions.

The following open-source software was directly used for this work:

PYTHON <https://www.python.org/>

ANACONDA <https://www.continuum.io/>

PYCHARM COMMUNITY EDITION <https://www.jetbrains.com/pycharm/>

SCILAB <http://www.scilab.org/>

VEUSZ <http://home.gna.org/veusz/>

CSV VIEWER <http://www.csvviewer.com/>

LATEX <https://www.latex-project.org/>

MIKTEX <http://miktex.org/>

TEXSTUDIO <http://texstudio.org/>

MINTED <https://github.com/gpoore/minted>

Additionally I would like to thank the people that pay taxes in Switzerland for coming up with a big chunk of the costs of my education. I hope to have it payed back someday.

I want to dedicate this thesis to all the people on the world that contribute to free software and free educational resources voluntarily.

My deepest regards to you.

For my personal well being throughout my bachelor's and master's studies I would like to thank the following people: Thierry Baasch, Damian Birchler, Remo Breitenmoser, Kerstin Cramer, Roman Ebneter, Arnold Engensperger, Denise Engensperger, Vanessa Engensperger, Valentin van Gemmeren, Sandro Kundert, Carine Neier, Tobias Peteler, Samuel Schafer, Franziska Schmidt, Sonja Segmüller, Rudolf Gerhard Stolz, Christoph Thormeyer, Thomas Witrahm, Jan Wolf, Sifu David Young, Christoph Zumbrunn

CONTENTS

1	INTRODUCTION	10
1.1	Motivation	10
2	STATE OF THE ART	11
2.1	Bone Tissue Engineering	11
2.1.1	Scaffold Designs	12
2.1.2	Material choices	13
2.2	Mechanics	14
2.2.1	Compliance	14
2.2.2	Meta-material	15
2.2.3	FEM	16
2.3	Optimization	17
2.3.1	Deterministic Optimization	17
2.3.2	Stochastic Optimization	18
3	GOALS	20
3.1	Mechanical Evaluation	20
3.2	Form Evaluation	21
3.3	Optimization	21
4	METHODS	22
4.1	Topologies	22
4.1.1	Topology connections	24
4.2	Finite Element Method	26
4.3	Mechanical Analysis	27
4.4	Optimization	29
4.4.1	Coupling of Design Variables	29
4.4.2	Optimization Problems	29
5	RESULTS	30
5.1	Topologies	30
5.1.1	Thickness Scaling	30
5.1.2	Cell Size Scaling	32
5.1.3	Ratio-changing Topologies	33
5.2	Optimization	36
5.2.1	Example of Minmaxing	36
5.2.2	Biomimetic Approach	38
6	DISCUSSION	41
6.1	Validity of Results	41
6.2	Optimization Performance	44
6.3	Possible Further work	44
6.3.1	Distortion of cells	44
6.3.2	Non-cubic cells	44

6.3.3	Combine Cell Topologies	44
6.3.4	Topology Optimization by Bendsøe and Kikuchi	44
6.3.5	Experimental comparison	44
6.3.6	Other scopes	45
7	CONCLUSION	46
	Appendices	47
A	APPENDIX	48
A.1	Topologies used in this thesis	48
A.2	Topologies from other research	50
A.3	Calculation for theoretical stiffness of meta-material	57
A.3.1	SciLab Code	57
A.4	Code	59
A.4.1	truss_builder.py	59
A.4.2	evaluation.py	66
A.4.3	Class_Script.py	72
A.4.4	Class_Cell.py	84
A.4.5	Class_Truss.py	85
A.4.6	library_cell.py	86
A.4.7	library_truss.py	103
A.4.8	csv_reader.py	105

LIST OF FIGURES

Figure 1	Cell Ratio Truncated Cubes	24
Figure 2	Cell Ratio Pyramids	24
Figure 3	Cell Ratio Truncated Octahedra	25
Figure 4	Face Diagonal Cubes Connection	25
Figure 5	FEM Beam Elements	26
Figure 6	Thickness Scaling	30
Figure 7	Close-up Thickness Scaling	31
Figure 8	Cell Size Scaling	32
Figure 9	Cell Ratio Scaling Truncated Cubes	33
Figure 10	Cell Ratio Scaling Truncated Octahedra	34
Figure 11	Cell Ratio Scaling Pyramids	34
Figure 12	Optimization Results MinMax	37
Figure 13	Optimization Results Biomimetic Approach $3 \times 3 \times 3$	39
Figure 14	Optimization Results Biomimetic Approach $5 \times 5 \times 5$	39
Figure 15	Sensitivity Analysis	41
Figure 16	Cutoff Error	42
Figure 17	Comparison of Beam and Solid Elements for the Cubes Topology	43
Figure 18	Comparison of Beam and Solid Elements for the Octets Topology	43

LIST OF TABLES

Table 1	Summarized properties that should be determined in the evaluation	21
Table 2	Topologies used in this thesis as a single cell and as a lattice consisting of three by three by three cells.	22
Table 3	The six states to calculate the compliance of a scaffold.	27
Table 4	Examples of the cell size scaling	32
Table 5	Optimization results for maximizing stiffness in z-axis and minimizing stiffness in x and y-axes.	36
Table 6	Optimization results for the biomimetic approach.	38
Table 7	Topologies used in this thesis as a single cell and as a lattice consisting of three by three by three cells.	48
Table 8	Topologies used in other research.	50

ACRONYMS

ETHZ	Eidgenössische Technische Hochschule Zürich — Swiss Federal Institute of Technology Zurich
EDAC	Engineering Design and Computing Lab
TE	Tissue Engineering
FEM	Finite Element Method
FEA	Finite Element Analysis
CAE	Computer-aided Engineering
CAD	Computer-aided Design
CAO	Computer-aided Optimization
DOA	Deterministic Optimization Algorithm
SOA	Stochastic Optimization Algorithm
AM	Additive Manufacturing
SLA	StereoLithography
SLM	Selective Laser Melting
SLS	Selective Laser Sintering
CG	Conjugate Gradient
BFGS	Broyden-Fletcher-Goldfarb-Shanno
L-BFGS	Limited Memory BFGS
L-BFGS-B	Limited Memory BFGS with bounds
SLSQP	Sequential Least Squares Quadratic Programming
HA	Hydroxyapatite
TCP	Tricalciumphosphate
PLA	Polylactic Acid

I

INTRODUCTION

1.1 MOTIVATION

Bone is a complex adaptive material. It decomposes and rebuilds itself throughout its life. It is of great importance to understand the mechanisms that cause bone to regrow to be able to develop new medical treatments . As of today, regrowing bone still poses a major challenge for large bone defects where the bone would not regrow by itself. [1]

Bone Tissue Engineering (TE) is a steadily developing, yet not fully understood method to regenerate large bone defects [2]. Small Scaffolds with sizes on the scale of centimeters are inserted into the bone, bridging the gap and providing the bone cells a place to grow and proliferate [3].

The scaffolds require pores to provide surfaces for bone cells to initially seed and form tissue. An approach from classical engineering would be to generate scaffolds consisting of beams to create a framework. The advantage of that design is the ability to evaluate the mechanical properties faster within a Finite Element Method (FEM) simulation by applying the Euler-Bernoulli beam theory instead of other methods.

Additive Manufacturing (AM) offers a method to create small scaffolds with controlled microscale features. This opens a new design space in terms of producibility which is not offered by many other methods.

The mechanical properties of the scaffold may also influence the time it takes to regrow bone. Bone has an intrinsic mechanism to reinforce its structure where stress is applied; Bone's mechanical properties also vary a lot throughout the body and depend on the direction force is applied [4]. Therefore it could be helpful to generate scaffolds that have the same compliance that bone has to avoid stress shielding ¹ and alleviate the regrowing phase for the body. By computationally optimizing the scaffold will also be possible to fit the mechanical properties of the individual bone.

This thesis shows an approach of automatically generating scaffolds as beam lattices with varying topology and beam thicknesses that are mechanically simulated and evaluated to fit certain target properties such as elastic modulus, shearing modulus, porosity and others to the properties of bone.

¹ Stress shielding: loads are taken by a prosthesis and shielded from going to the bone [5]

2

STATE OF THE ART

2.1 BONE TISSUE ENGINEERING

Bone-TE has the goal to surgically repair or replace damaged bone by implanting cells, scaffolds, DNA, protein or protein fragments.[1] There are three widely accepted approaches [6],[7], [8]:

1. Implant a scaffold for cells to grow and proliferate
2. Extract cells and cultivate them in vitro to transplant them back into the patient.
3. Extract cells and cultivate them in a scaffold in vitro to transplant the cell-scaffold construct back into the patient.

The method introduced in this thesis is based on the first approach.

Main cases to use Bone-TE on a human are for spinal fusion¹ or for major fractures.

The developed scaffold should have a certain **stiffness** that is not too soft, which could cause the structure to collapse, and not too hard, which could cause stress shielding that would slow down the rehabilitation process. One can argue that the target compliance of the meta material should be as close as possible to actual bone, also known as the biomimetic approach. The biomimetic approach requires a material with higher elastic modulus than polymers, reducing the number of material choices. On the other hand it is common to add a support structure to keep the bones and the scaffold in place during rehabilitation which takes a lot of load away from the scaffold, making its mechanical properties require a different tuning.

For Bone Tissue to regrow, a widely used method is to focus on the **pore size** of a potential scaffold. If the pore size of a scaffold is too big it will take a longer time for the bone tissue to grow. If the pore size is too small the scaffold will lack nutrient transport.

Coming along with pore size is **porosity**. Tendency wise one could say that the higher the porosity, the better the nutrient flow, but the lower the stiffness of the material. However, the interconnectivity of cells is a mechanic that is

¹ Spinal fusion: A surgical technique to fuse two vertebrae. [9]

more complex and requires fluid dynamics simulations to provide a well researched statement about nutrient flow.

The most important factor is the biocompatibility. Most materials cause various foreign body reactions such as inflammation or even necrosis. This limits the material choices that can be made for engineered tissue. Another important quality of a bone-TE-scaffold is **absorbability**. Ideally the scaffold gets absorbed in the process leaving nothing but the regrown bone tissue. It is very difficult to achieve the right absorbability, because the scaffold must not be absorbed too fast nor too slow.

In summary taken from [10], desirable qualities of a bone-TE scaffold are the following

- Available to surgeon on short notice
- Promotes bone ingrowth
- Absorbs in predictable manner in concert with bone growth
- Does not induce soft tissue growth at bone/implant interface
- Adaptable to irregular wound site, malleable
- Average pore sizes approximately 200-400 μm
- Maximal bone growth through osteoinduction and/or osteoconduction
- No detrimental effects to surrounding tissue due to processing
- Correct mechanical and physical properties for application
- Sterilizable without loss of properties
- Good bony apposition
- Absorbable with biocompatible components

2.1.1 *Scaffold Designs*

Scaffold designs can be divided into two sections:

Discrete scaffolds that consist only of struts/beams/bars creating a lattice structure, and **continuous scaffolds** which are made by adding or subtracting volumetric shapes. While the discrete scaffolds offer a way for faster evaluation (see 2.2.3), the continuous scaffolds allow more intricate designs. A few designs from other research are shown in Appendix A.2 to give an impression of the vast amount of design possibilities.

2.1.2 Material choices

Important aspects regarding the material choice are **biocompatibility**, **absorbability** and **mechanical properties**. Research can be found about the following materials ([11], [12]):

Metals

The high stiffness of **Titanium** (Young's modulus of about 125 GPa) and its alloys allows very thin struts and high porosity while still achieving the stiffness of bone. The human body does not absorb those metals and the scaffold will stay inside the patient which can cause stress shielding [13]. A promising material choice may be **Magnesium** (Young's modulus of about 46 GPa), which is absorbable by corrosion [14]. While a scaffold made of an alloy is usually manufactured via the additive manufacturing method Selective Laser Sintering (SLS), metallic foams can be created by a powder metallurgical process [15].

Ceramics

Bone itself consists of at least 50% Hydroxyapatite (HA) [1] (ref Paper optimal design and fabrication...), which makes HA a logical choice for bone tissue engineering. Another ceramic that has been used is α - β -Tricalciumphosphate (TCP) (ref Paper and weber).

Polymers

Examples of biocompatible polymers [16] which can be manufactured additively are **Polycaprolactone** (PCL) [17], **Polylactic acid** (PLA) [18] or **Polypropylene fumarate/diethyl fumarate** (PPF/DEF) [19].

Others

Other approaches include the materials **bioglass biocomposites** [20], **nHAC powder/PLA composite** [18], [21]. Both materials are bioactive ².

² A bioactive material causes a biological response at the interface of the material which results in a bond between the tissue and the material [22]

2.2 MECHANICS

2.2.1 Compliance

The compliance offers a useful way to describe a structure's mechanical properties. It can be described by a matrix and contains the information that characterizes the linear elastic response of the structure to applied forces (see equation 1). From the compliance matrix properties such as Young's modulus, shearing modulus and Poisson's ratio can be calculated (see eq. 5). The compliance is independent of the outer form of a structure, which is ideal for describing a scaffold which might be repeated continuously. The compliance matrix also contains the information whether a material is anisotropic (eq. 2), orthotropic (eq. 4) or isotropic (eq. 3).

The relation between strain and stress in a material can be described by the compliance matrix.

$$\begin{pmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \gamma_{23} \\ \gamma_{31} \\ \gamma_{12} \end{pmatrix} = \epsilon = C * \sigma = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} \end{pmatrix} * \begin{pmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \tau_{23} \\ \tau_{31} \\ \tau_{12} \end{pmatrix} \quad (1)$$

Anisotropic Compliance is characterized by non-zero elements in the top right and bottom left quadrant of the compliance matrix. This means, that the material reacts with shearing displacements on compression or vice-versa.

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} \end{pmatrix} \quad (2)$$

An Isotropic Compliance describes a material which is uniform in all directions.

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{12} & 0 & 0 & 0 \\ c_{12} & c_{11} & c_{12} & 0 & 0 & 0 \\ c_{12} & c_{12} & c_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{44} & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{44} \end{pmatrix}; c_{44} = c_{11} - c_{12} \quad (3)$$

Orthotropic Compliance, where the matrix is symmetric and its top right and bottom left quadrant is zero as well as the bottom right quadrant is a diagonal matrix.

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & 0 & 0 & 0 \\ c_{21} & c_{22} & c_{23} & 0 & 0 & 0 \\ c_{31} & c_{32} & c_{33} & 0 & 0 & 0 \\ 0 & 0 & 0 & c_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{55} & 0 \\ 0 & 0 & 0 & 0 & 0 & c_{66} \end{pmatrix} \quad (4)$$

These equations show the connection between compliance and other material properties for an orthotropic material.

$$\begin{aligned} E_{11} &= \frac{1}{c_{11}} & E_{22} &= \frac{1}{c_{22}} & E_{33} &= \frac{1}{c_{33}} \\ \nu_{12} &= -\frac{c_{12}}{c_{11}} & \nu_{23} &= -\frac{c_{23}}{c_{22}} & \nu_{31} &= -\frac{c_{31}}{c_{33}} \\ \nu_{21} &= -\frac{c_{21}}{c_{22}} & \nu_{32} &= -\frac{c_{32}}{c_{33}} & \nu_{13} &= -\frac{c_{13}}{c_{11}} \\ G_{23} &= \frac{1}{c_{44}} & G_{31} &= \frac{1}{c_{55}} & G_{12} &= \frac{1}{c_{66}} \end{aligned} \quad (5)$$

$$\begin{aligned} E_{xx} &: \text{Elastic Modulus} \\ \nu_{xy} &: \text{Poisson's ratio} \\ G_{xy} &: \text{Shearing Modulus} \end{aligned}$$

2.2.2 Meta-material

A meta-material is an arrangement of engineered elements, where the local properties differ from the global properties. This is used to create specific physical properties. One could also say, that meta-materials present the next level of structural organizational matter [23].

In this thesis, the generated scaffolds are regarded as meta-materials. The

meta-material does not take account of the intrinsic truss structure, which allows for describing the compliance (see 2.2.1) as if it were a genuine material. This is particularly useful for scaffold applications, since the outer measurements of the scaffold may change for the individual use, whereas the local structure stays the same.

2.2.3 *FEM*

The FEM (or Finite Element Analysis (FEA)) subdivides a larger problem of partial differential equations into numerical subproblems of sets of linear equations. For a mechanical analysis this means that the approach from continuum mechanics is subdivided into discrete elements. The sets of linear equations can be solved numerically by a computing system leading to an approximation of the solution. There are different FEM-solvers commercially available. For this thesis the ABAQUS-solver by SIMULIA was used.

2.3 OPTIMIZATION

Mathematical optimization (alternatively mathematical programming) is the method of searching for the best set of variables to minimize the value of a function. Usually one considers a problem as seen in equation 6. An algorithm is defined that evaluates the function for certain sets of variables and follows a strategy to find sets which minimize or maximize the value of the function (based on the function evaluations).

It is important to note, that there is a diversity of algorithm types that differ in their effectiveness in searching a problem space to find best sets in the shortest time for any given problem. The problem needs to be analyzed and a fitting algorithm has to be found. Optimization can be further divided into deterministic (2.3.1) and stochastic (2.3.2) methods.

$$\begin{aligned}
 \min_x \quad & f(x) = w_1 E_1(x) + w_2 E_2(x) + \dots \\
 \text{s.t.} \quad & g_1(x) \leq 0 \\
 & g_2(x) \leq 0 \\
 & \vdots \\
 & h_1(x) = 0 \\
 & h_2(x) = 0 \\
 & \vdots
 \end{aligned} \tag{6}$$

- $f(x)$: Objective Function
- $E_n(x)$: Evaluation functions
- w_n : Weighting Factor for E_n
- $g_n(x)$: Inequality constraint
- $h_n(x)$: Equality Constraint

2.3.1 Deterministic Optimization

The main idea behind all deterministic methods is to search for the direction of steepest descent of the evaluation function and perform a line search into that direction until the set of variables for the smallest function evaluation is found. After that, this process is repeated until a stopping criteria is fulfilled. Deterministic Optimization Algorithm (DOA) search for the solution without the use of randomness. This means that if the initial set of variables is the same the final set of variables will be the same as well.

DOAs can be further classified by the amount of information they require to find the next search direction. The simplest DOAs sustain solely on the evaluation of the function itself (f.e. Downhill Simplex, Nelder-Mead, Powell, Response Surface). More extensive algorithms request the evaluation of the first derivatives with respect to the optimization variables, which is also called

$$H = \begin{pmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{pmatrix} \quad (8)$$

It is evident, that for a large set of variables the Hessian becomes a greater calculation effort.

the gradient of the function (equation 7), a vector that shows towards the direction of steepest descent (f.e. Cauchy, Fletcher-Reeves). The most expensive algorithms calculate the second derivatives, a matrix called the Hessian (equation 8) (f.e. Newton, Quasi-Newton, BFGS) [24].

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix} \quad (7)$$

2.3.1.1 The BFGS-Algorithm

The Broyden-Fletcher-Goldfarb-Shanno (BFGS)-algorithm is an approximate of the Newton's method. Instead of evaluating the Hessian matrix of second derivatives, an approximation of it is updated every step by using the gradient information. The BFGS-Method is one of the most popular members of this class [24]. Also quite common is the Limited Memory BFGS (L-BFGS)-method, which differs from BFGS by having a limited memory usage, and the Limited Memory BFGS with bounds (L-BFGS-B)-method, which additionally to limited memory makes use of bounding constraints. L-BFGS-B is implemented into the *optimize*-package of the *scipy*-package, which is a open-source software for mathematics, science and engineering with the programming language *python*.

2.3.2 Stochastic Optimization

In contrast to deterministic algorithms (2.3.1), a Stochastic Optimization Algorithm (SOA) does not always have the same outcome each time it is run. New sets of variables are determined with a random aspect. Rather than searching for a direction with a descent of the function value, SOAs define new sets of variables randomly. More complicated algorithms use the information they get to search in areas that are more likely to have better solutions.

A common group of SOAs are called evolutionary algorithms which are inspired by the evolution of biological organisms. These algorithms initially evaluate the objective function with random sets of variables. They then proceed to search for new solutions close to the sets which showed better results. Other prominent algorithms are Random Search, Stochastic Hill Climbing, Simulated Annealing.

3

GOALS

The objective of this thesis can be divided into 3 points:

MECHANICAL EVALUATION Computationally evaluate mechanical properties of a given scaffold

FORM EVALUATION Computationally evaluate form properties of a given scaffold

OPTIMIZATION Computationally optimize for target values respective to mechanical and form properties

3.1 MECHANICAL EVALUATION

The evaluation should be able to determine the reaction of the scaffold to compression and shearing loads. From the displacements of the framework caused by applied loads the compliance can be calculated to be able to fully describe the mechanical behaviour of the scaffold. The compliance matrix describes the mechanical properties of the meta material (see 2.2.1 and 2.2.2). By only using struts the whole scaffold can be evaluated in a FEM software with a wire frame that uses the Bernoulli beam theory to calculate the mechanical reaction to an applied load. The properties that should be evaluated are **Young's modulus**, **shearing modulus** and **Poisson's ratio** which can all be read out of the compliance.

3.2 FORM EVALUATION

An interesting form property can be described as the **porosity** of the meta material which is defined by the volume of the scaffold divided by the volume of the whole meta material. Some research refers to the **void ratio** which is described by equation 9.

Other form properties that are of interest would be the **surface area** and the **pore size** (As mentioned in 2.1).

$$e = \frac{\phi}{1 - \phi} \quad (9)$$

e : Void Ratio
 ϕ : Porosity

3.3 OPTIMIZATION

Having the computer automatically evaluate the mechanic and form properties of a scaffold allows to implement an automated computational optimization. The goal would be to determine interesting solutions within an acceptable amount of time. Once adequate solutions were found for certain target values, the other properties can be further compared between the topologies.

Mechanical Properties	Form Properties
Young's Modulus	Volume
Shearing Modulus	Porosity/Void Ratio
Poisson's ratio	Surface Area
Compliance Matrix	Pore Size

Table 1.: Summarized properties that should be determined in the evaluation

4

METHODS

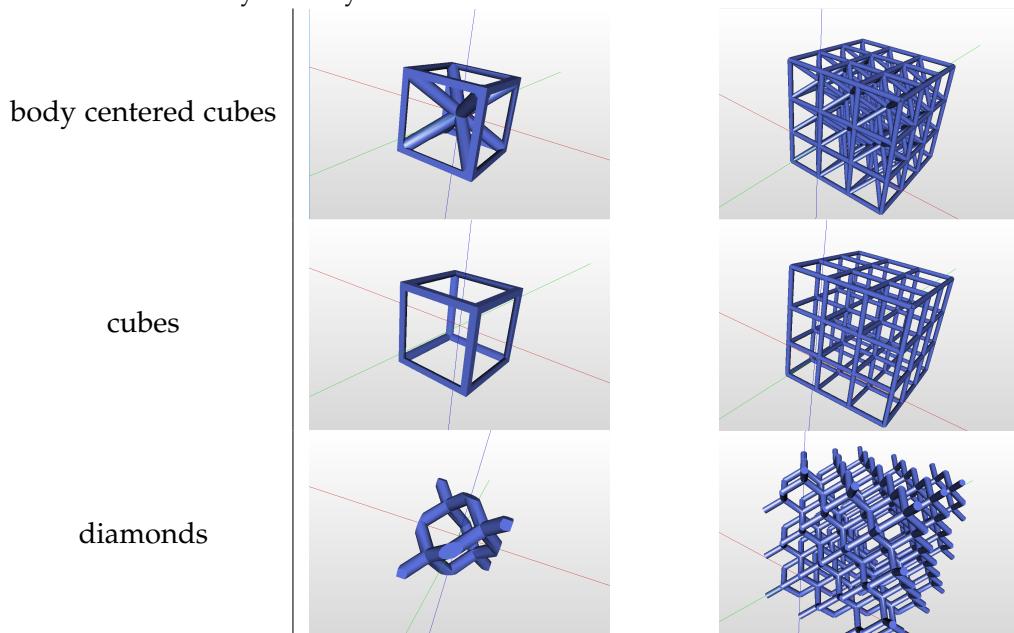
4.1 TOPOLOGIES

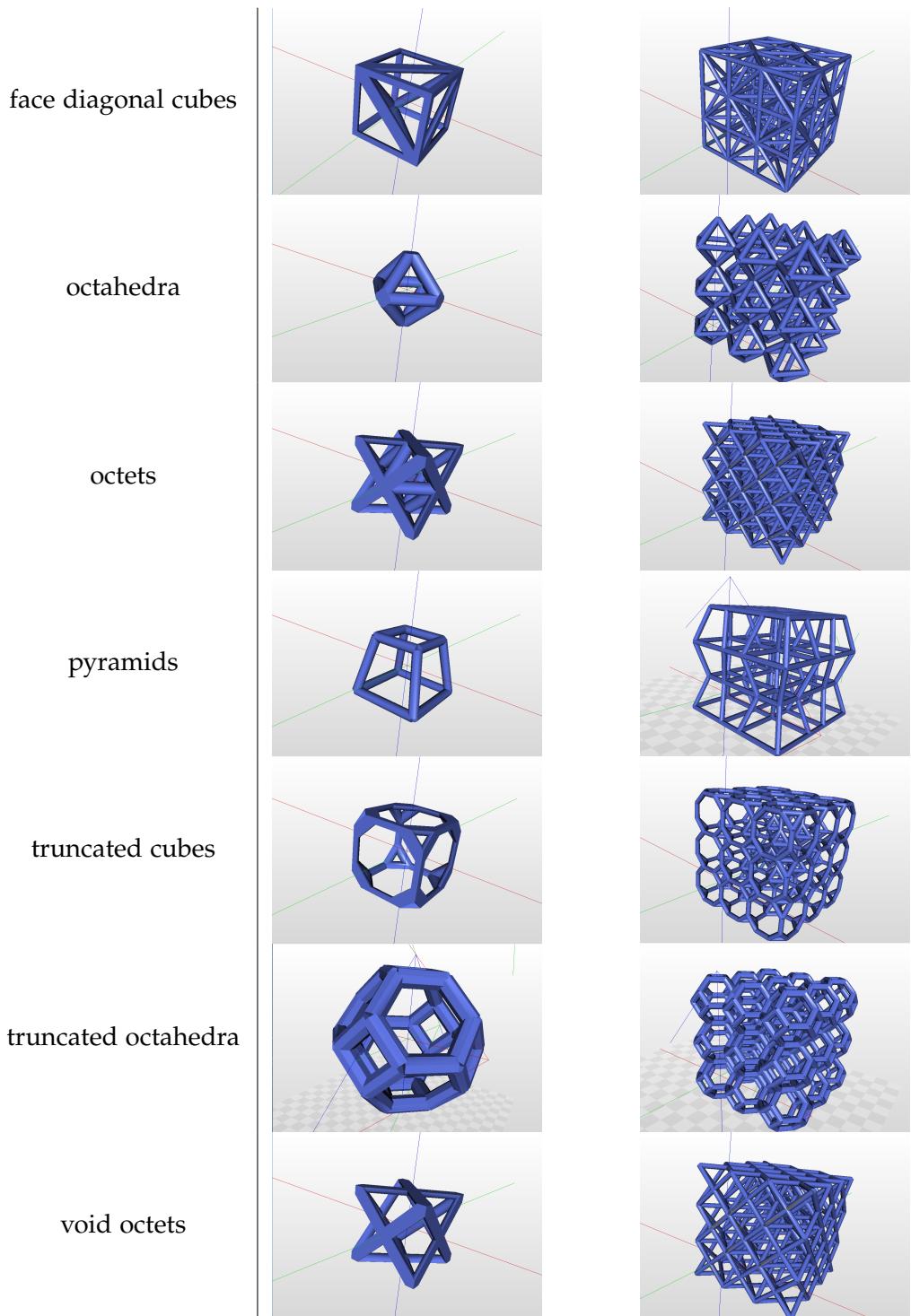
The unit cell topologies of the scaffolds are not automatically designed, but were developed by hand (see Table 2) and then automated to form a repeating solid structure, for form evaluation (3.2), and wire frame, for mechanical evaluation (3.1). Important factors for the design were inspiration by nature and geometry as well as comparability to other research. A general concern was to keep the topologies simple if possible to reduce the number of design variables required to tune each unit cell.

Because of the simplicity of **cubes** (see Table 2, Column 2) and its presence in other research ([30], [31], [32], [37], [38]) this model was chosen for comparison and validation of results.

As can be seen in the three by three by three lattices in Table 2, the unit cells are combined at their faces, which causes struts which are along the face of a unit cell to fuse with the strut from the unit cell next to it.

Table 2.: Topologies used in this thesis as a single cell and as a lattice consisting of three by three by three cells.





4.1.1 Topology connections

Three of the evaluated topologies, namely **truncated cubes** (Figure 1), **pyramids** (Figure 2), and **truncated octahedra** (Figure 3) allowed for a continuous change of the topology depending on a single variable which was called the cell ratio. An example of this ratio is the amount of truncation that is applied on the cubes topology. By reducing the truncation to zero, the cubes topology is created, and by maxing out the truncation, the original struts disappear resulting in the cuboctahedra topology (see Figure 1). In theory the cuboctahedra topology is equal to the octahedra topology in terms of strut arrangement. The cuboctahedra topology is the octahedra topology translated by half a cell size into every direction.

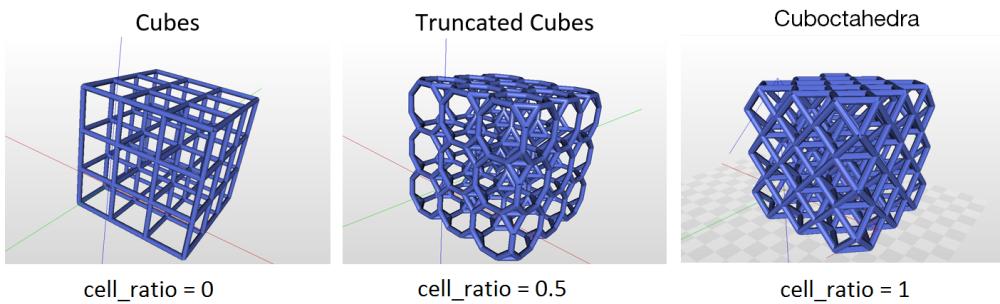


Figure 1.: This figure depicts the dependence of the truncated cubes topology to the cell ratio variable.

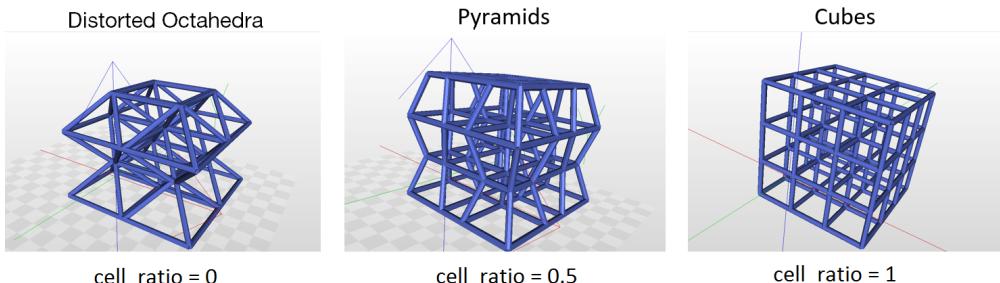


Figure 2.: This figure depicts the dependence of the pyramids topology to the cell ratio variable.

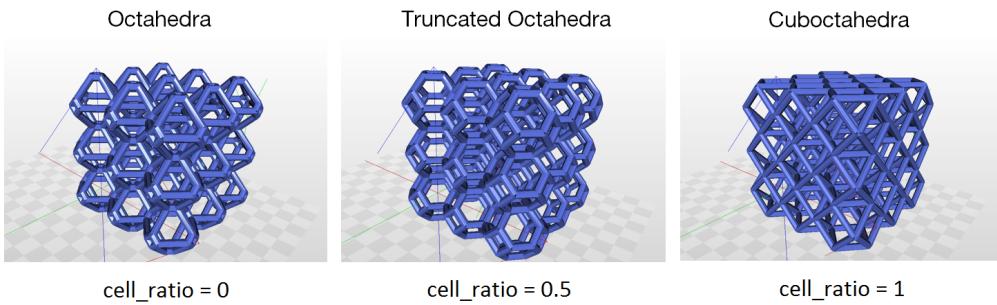


Figure 3.: This figure depicts the dependence of the truncated octahedra topology to the cell ratio variable. The octahedra topology on the right is shifted by half a cell in comparison to the octahedra topology on the left

Another observation that was made is that the face diagonal cubes topology is the combination of the octet topology and the cubes topology, as is made clear in Figure 4.

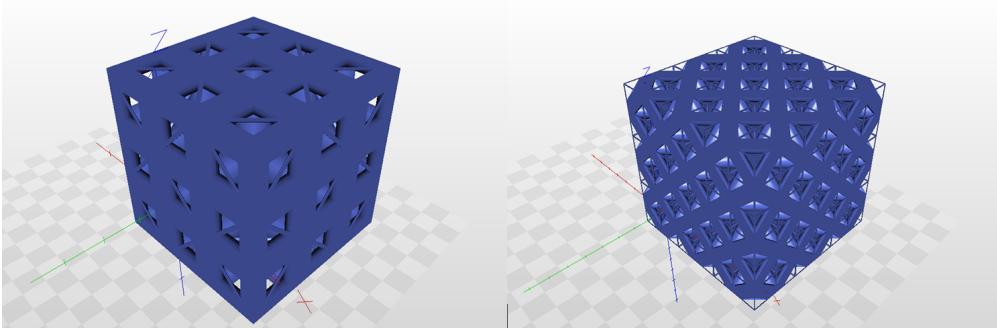


Figure 4.: On the left the face diagonal cubes topology is shown with very thin diagonal struts creating the cubes topology. On the right the same topology with doubled amount of cells per side is shown with very thin straight struts resulting in the octets topology.

4.2 FINITE ELEMENT METHOD

To evaluate the scaffold mechanically the FEM(see 2.2.3) was used. The mechanical behaviour of each strut of the scaffold is approximated with the euler bernoulli beam theorem. Quadratic shape functions were used while the element's length is set at a third of the single cell length (see Figure 5) in order to provide sufficient mechanical behavior. All evaluations are based on a linear elastic behaviour.

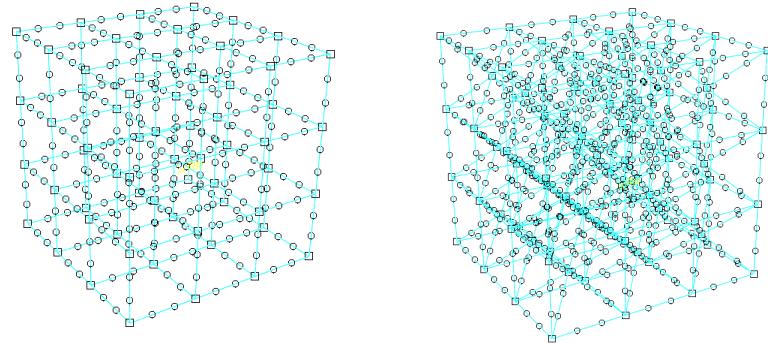
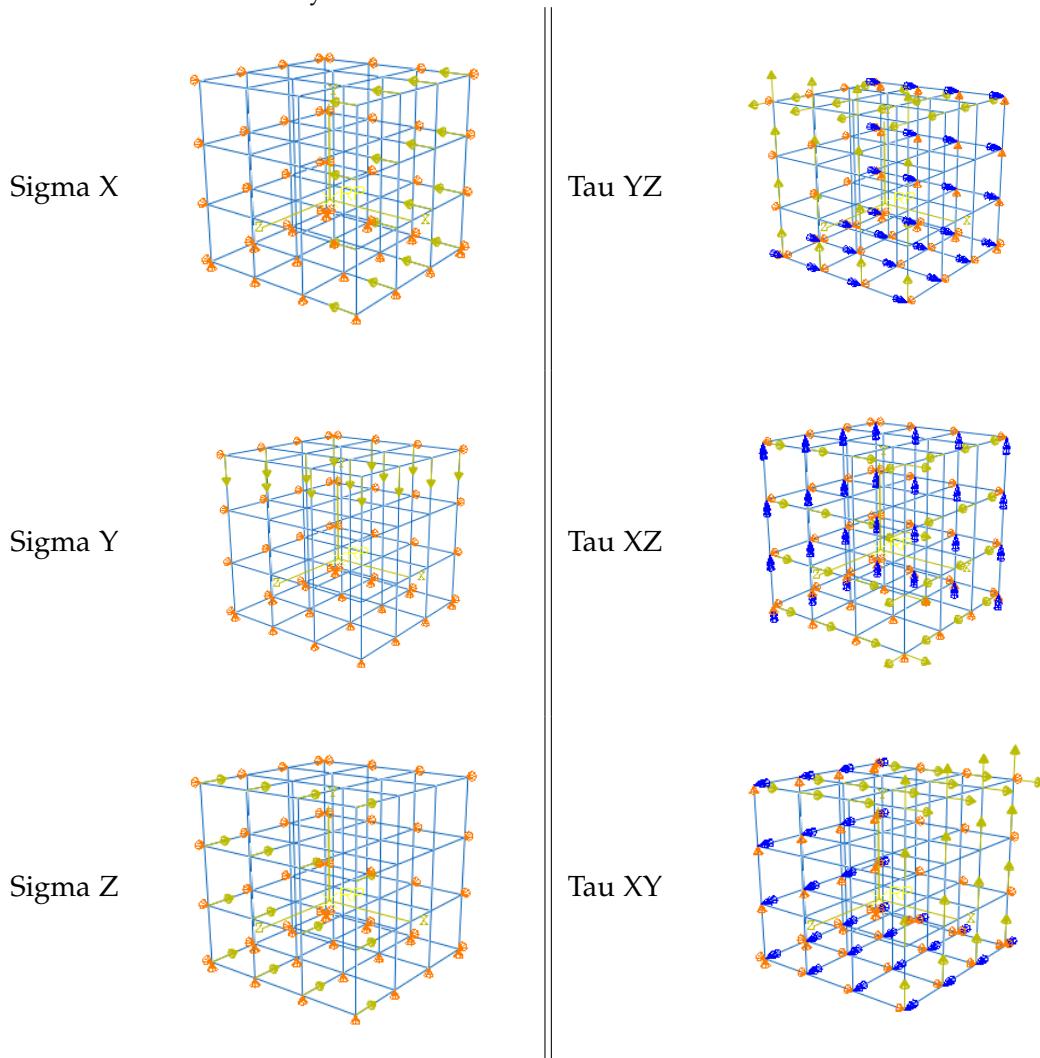


Figure 5.: FEM beam elements for three by three by three scaffolds with the **cubes** and **body centered cubes** topologies.

4.3 MECHANICAL ANALYSIS

To fully mechanically evaluate an orthotropic scaffold, six separate states are needed: 3 steps for determining the elastic moduli and 3 steps to determine the shearing moduli. In each state loads and fixations are applied to match a loading condition (see 3). The medians of the displacements on each side of the scaffold are read out to calculate the elements of the compliance (see 2.2.1) of the meta material.

Table 3.: The six states to calculate the compliance of a scaffold. The yellow arrows depict concentrated forces, the orange tips represent translational fixations and the blue symbols stand for rotational fixations



$$\begin{pmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \gamma_{23} \\ \gamma_{31} \\ \gamma_{12} \end{pmatrix} = \boldsymbol{\epsilon} = C * \boldsymbol{\sigma} = \begin{pmatrix} * & c_{12} & * & * & * & * \\ * & c_{22} & * & * & * & * \\ * & c_{32} & * & * & * & * \\ * & c_{42} & * & * & * & * \\ * & c_{52} & * & * & * & * \\ * & c_{62} & * & * & * & * \end{pmatrix} * \begin{pmatrix} 0 \\ \sigma_{22} \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad (10)$$

$$d_{11}/l_{scaffold} = \epsilon_{11} = c_{12} * \sigma_{22} = c_{12} * \frac{F}{A_{scaffold}} \longleftrightarrow c_{12} = \frac{d_{11} * A_{scaffold}}{F * l_{scaffold}} \quad (11)$$

- d_{11} : displacement
 $l_{scaffold}$: length of the scaffold
 F : applied force
 $A_{scaffold}$: area of the scaffold

These equations show an example of the data that can be extracted from one simulation state (in this case **Sigma Y** and c_{12}). For each simulation state, six elements of the compliance matrix can be determined.

4.4 OPTIMIZATION

4.4.1 Coupling of Design Variables

To reduce the number of design variables and therefore reduce the search space, the thicknesses of certain struts were coupled. By decreasing the design variables, the optimization will find solutions exponentially faster. Nevertheless, there still need to be solutions that fit the optimization goal. The following rules were applied:

1. **The strut vis-à-vis** of the cell is coupled because it has to be equal if the design gets repeated continuously (as mentioned in 4.1).
2. All struts that face the **same direction** are coupled.
3. To always generate orthotropic materials **asymmetry is avoided**.

4.4.1.1 Design Space Characterization

Because of the way the design variables have been chosen it is to assume that the design space is convex. If the thickness of a strut that faces a certain direction increases, the whole stiffness of the meta material most likely increases as well in that same direction of the strut.

A deterministic algorithm, namely the BFGS-method (see 2.3.1.1) was used.

4.4.2 Optimization Problems

To show the capabilities of the method of this thesis, the following optimization goals are set:

MaxMin Optimization

Maximize for elastic modulus in z-direction and minimize for elastic modulus in x- and y-direction. The design variables are the thicknesses of the struts (see 5.2.1).

Biomimetic Optimization

Fit the elastic modulus properties to those of human cortical bone, for a scaffold made of titanium. The design variables are the thicknesses of the struts (see 5.2.2).

5

RESULTS

5.1 TOPOLOGIES

5.1.1 Thickness Scaling

In Figure 6 the evaluation for different topologies is shown with increasing thickness of the scaffold struts while keeping the cell size constant. Figure 7 zooms into the area with porosity of 60% to 90%, which is the interesting area to bone tissue engineering research for nutrient flow and space for cells to bind and proliferate on (see 2.1). Additionally, as will be mentioned in 6.1, the results are less accurate for lower porosities.

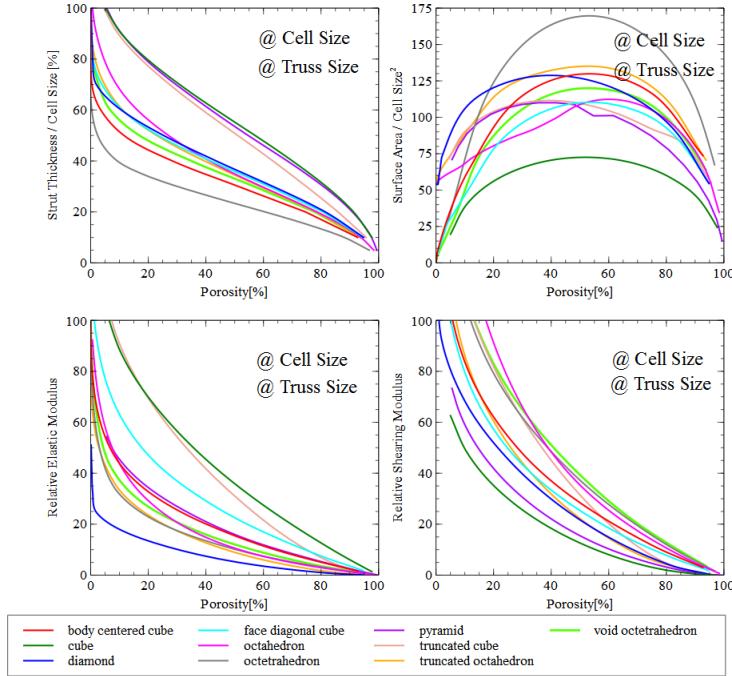


Figure 6.: Simulation results for different topologies in respect to porosity

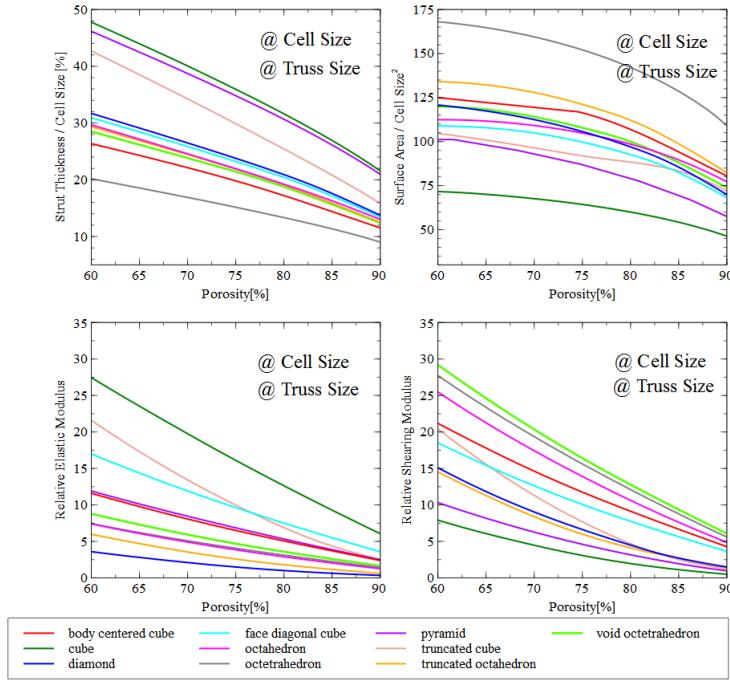


Figure 7.: Close-up of simulation results for different topologies in respect to porosity

Interpretation

As can be seen in other research ([26]), a distinction can be made between stretch-dominated and bend-dominated topologies. Stretch-dominated structures, such as cubes, body centered cubes, face diagonal cubes or octets, react to loads with axial stress in the struts, whereas bend-dominated structures, such as truncated octahedrons, octahedrons, diamonds or void octets, react with bending.

It can be seen in Figure 7 of section 5.1.1 that topologies that excel in compression generally perform bad in shearing and the other way around. One could say, that the no free lunch theorem applies here.

Some topologies perform worse than others in both cases. F.e. the diamond topology is outperformed by all topologies in compression and has a lower shearing modulus than most topologies. Because of the scope of this work no topology can really be excluded, since the mechanical properties might have to be low for some material choices.

5.1.2 Cell Size Scaling

In theory, the wanted surface area or pore size can be adjusted in the scaffold by scaling cell size and strut thickness proportionally while the other properties remain constant (see A.3). To compare this phenomenon, the topologies were evaluated at the same overall truss size, but with increasing number of cells and therefore decreasing cell size (see Figure 8). It is due to the inaccuracies that are mentioned in 6.1, that especially the elastic modulus deviates.

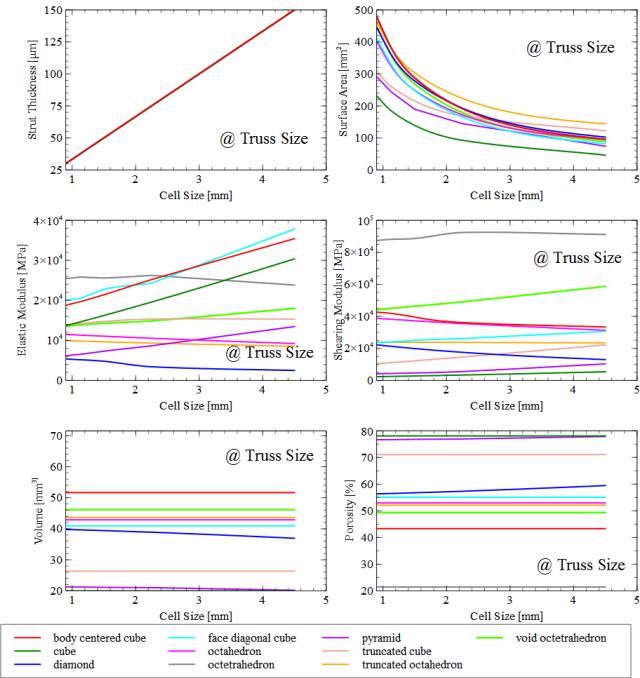


Figure 8.: Simulation results for different topologies while keeping the ratio between cell size and strut thickness constant.

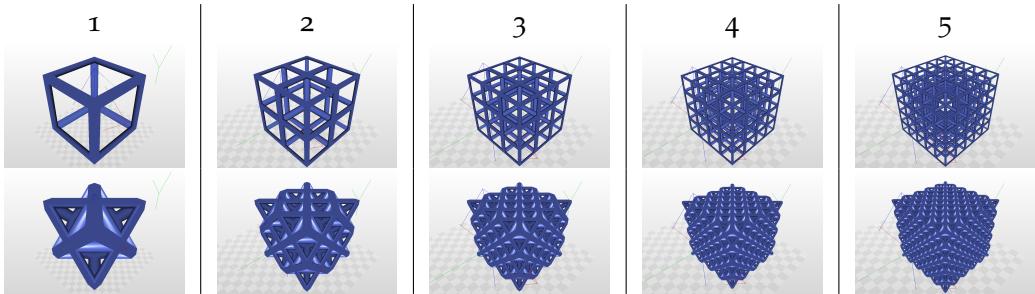


Table 4.: Examples of the cell size scaling, where the number of cells increases, but the overall truss size and porosity stays constant. This would allow for tuning the surface area and pore size of a given scaffold without changing its other properties.

5.1.3 Ratio-changing Topologies

As mentioned in 4.1.1, three topologies were further observed with a variable called the cell ratio that could change the topology of the cell f.e. the truncation of the cubes in the truncated cubes topology. In Figures 9, 10 and 11, the thickness scalings are shown with gradually increasing cell ratio.

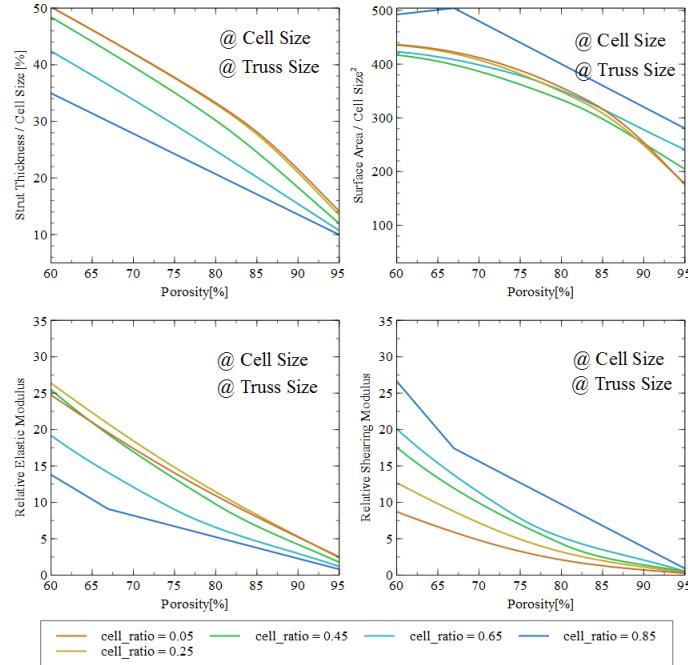


Figure 9.: Simulation results for different cell ratios for the truncated cubes topology.

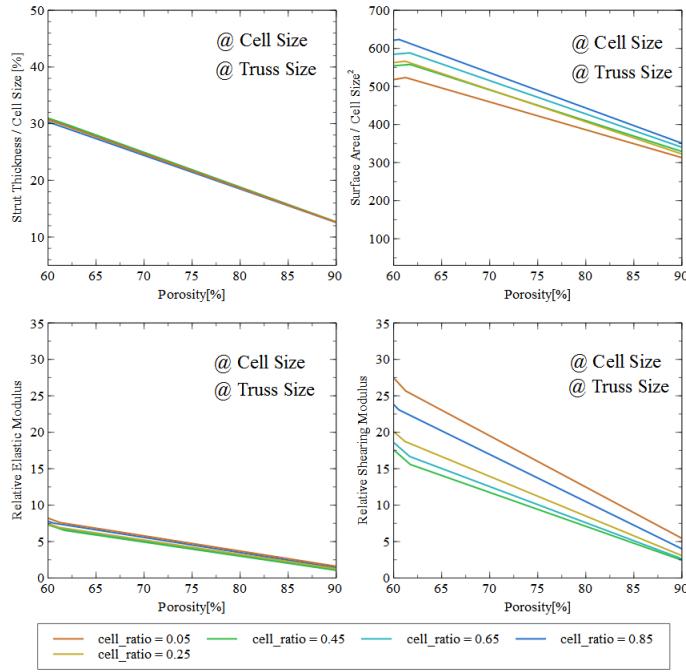


Figure 10.: Simulation results for different cell ratios for the truncated octahedra topology.

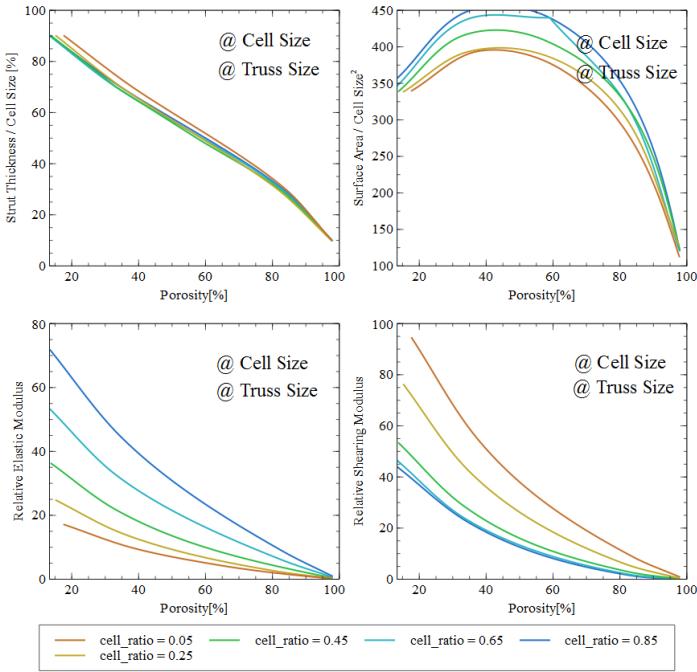


Figure 11.: Simulation results for different cell ratios for the pyramids topology.

Interpretation

These types of topologies offer a simple way of continuously varying the topology of a scaffold which might be an interesting optimization variable to use. Especially interesting are the pyramids and the truncated cubes topologies since their mechanical properties changes from high elastic modulus and low shearing modulus to low elastic modulus and high shearing modulus. The truncated octahedra and pyramids keep a nearly constant strut thickness over the cell ratio change at a given porosity, which might also be a desirable characteristic.

5.2 OPTIMIZATION

5.2.1 Example of Minmaxing

To test the capabilities of the presented method, an optimization goal was set that allowed for visual approving. Therefore the optimization goal was set to maximize stiffness into z-direction and minimize stiffness into x- and y-direction (see equation 12).

$$\begin{aligned} \min_x \quad & f(x) = E_x(x) + E_y(x) - E_z(x) \\ \text{s.t.} \quad & g_1(x) = x_1 \geq 0 \\ & g_2(x) = x_2 \geq 0 \\ & \vdots \end{aligned} \tag{12}$$

$f(x)$: Objective Function

$g_n(x)$: Inequality constraint

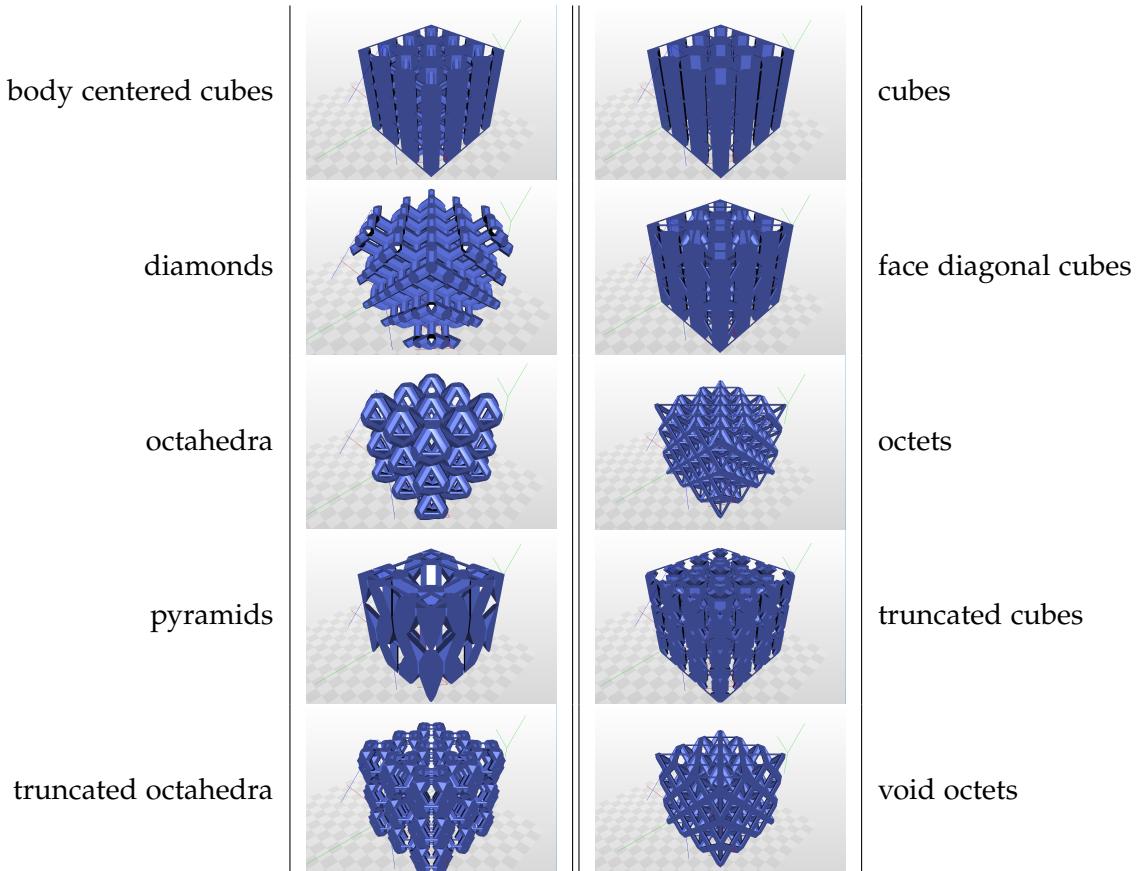


Table 5.: Optimization results for maximizing stiffness in z-axis and minimizing stiffness in x and y-axes.

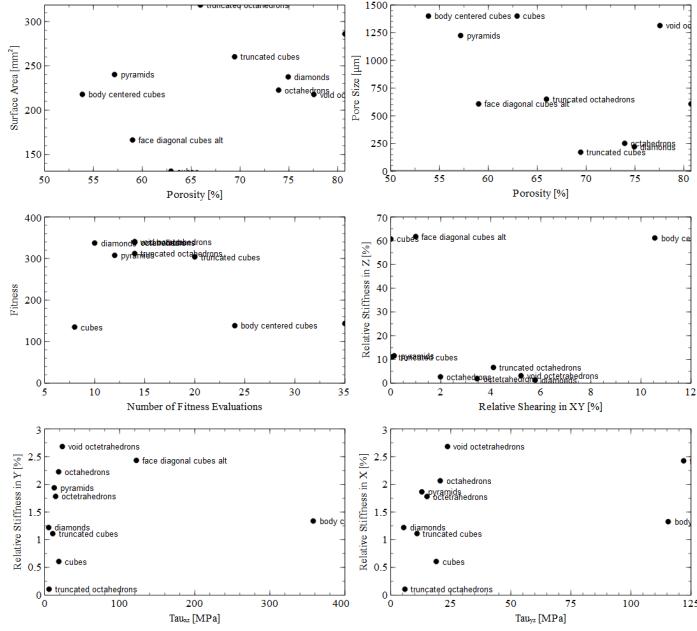


Figure 12.: Optimization results for the MinMax example with three by three by three cells.

Interpretation

As can be seen in Table 5, the optimization resulted in increasing the thickness of vertical struts and decreasing the thickness of the horizontal struts, as is expected. This task was especially well solved by the cubes, body centered cubes and face diagonal cubes topologies. In Figure 12 the properties of the resulted scaffolds for all topologies are depicted. This provides a multidimensional pareto front for interesting properties from which a topology can be chosen.

5.2.2 Biomimetic Approach

The optimization goal for the biomimetic approach was set to fit the elastic modulus of the meta-material to the elastic modulus of human cortical bone, namely 15 GPa into longitudinal and 12 GPa into transverse direction [25].

$$\begin{aligned}
 \min_x \quad & f(x) = \frac{1}{300} |E_x(x) - 12\text{GPa}| + \frac{1}{300} |E_y(x) - 12\text{GPa}| + \frac{1}{300} |E_z(x) - 15\text{GPa}| \\
 \text{s.t.} \quad & g_1(x) = x_1 \geq 0 \\
 & g_2(x) = x_2 \geq 0 \\
 & \vdots \\
 & g_n(x) \geq 0
 \end{aligned} \tag{13}$$

$f(x)$: Objective Function
 $g_n(x)$: Inequality constraint

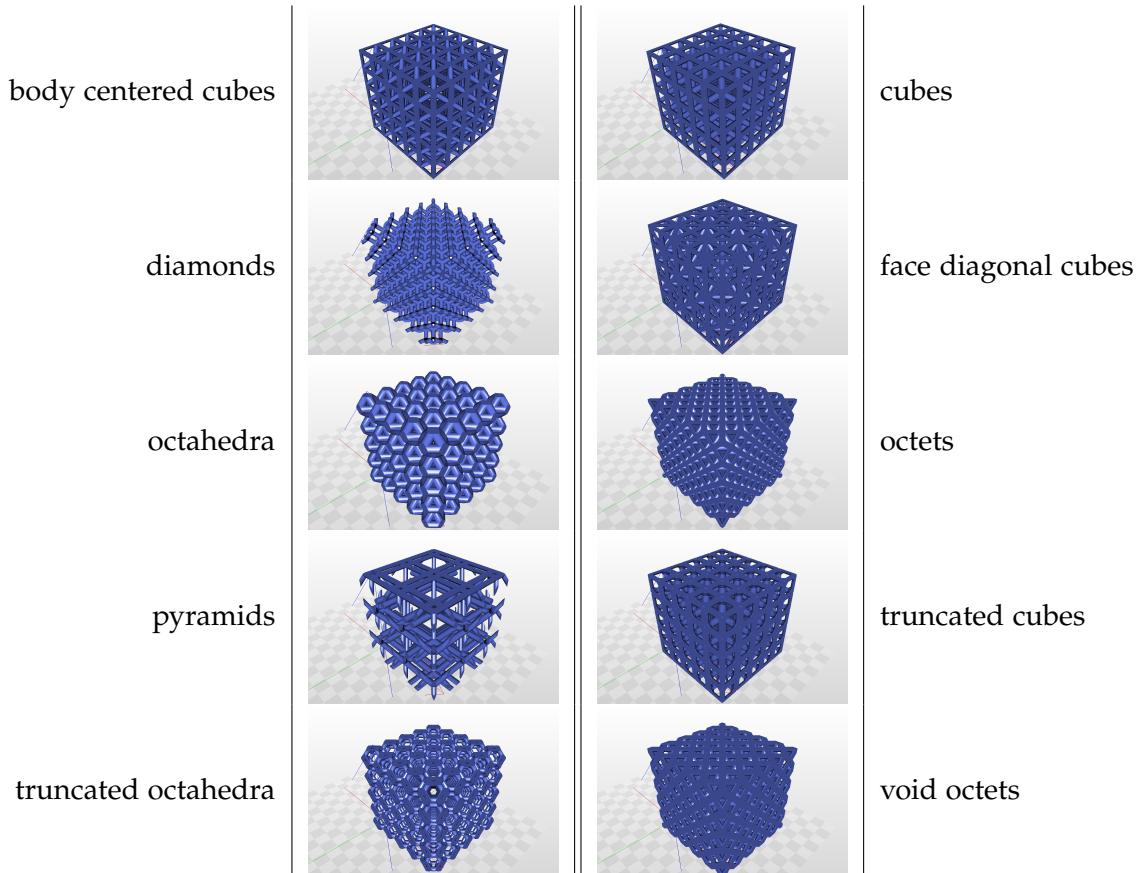


Table 6.: Optimization results for the biomimetic approach.

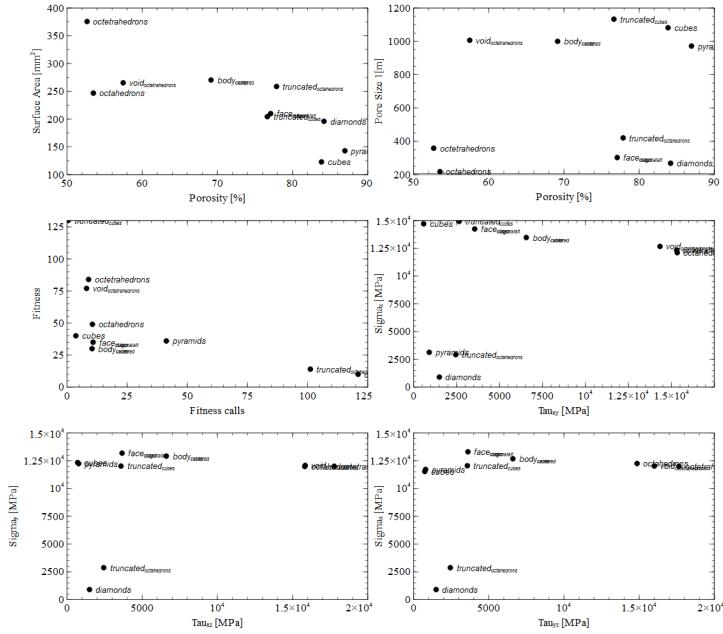


Figure 13.: Optimization results for the biomimetic approach with three by three by three cells.

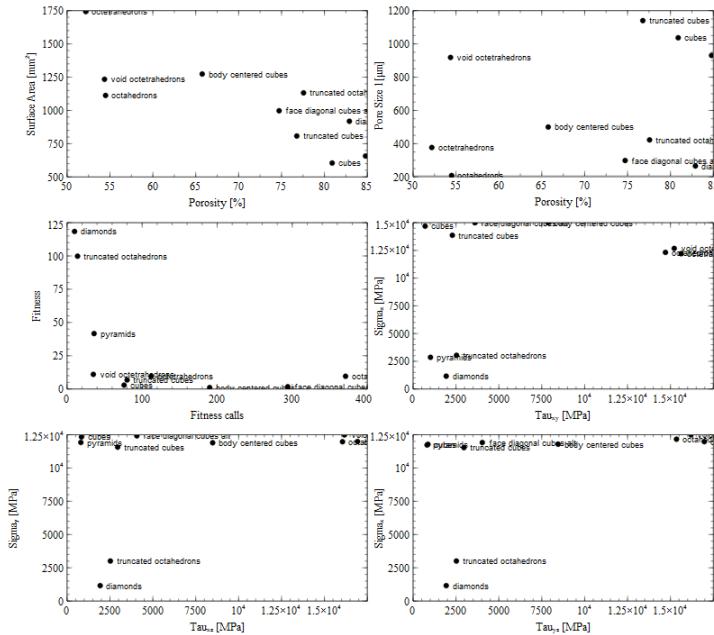


Figure 14.: Optimization results for the biomimetic approach with five by five by five cells.

Interpretation

In Figure 13 and 14 the properties of the resulted scaffolds for all topologies are depicted. As already mentioned in the interpretation of 5.2.1, this provides a multidimensional pareto front for interesting properties from which a topology can be chosen. The resulting scaffolds can be seen in Table 6.

6

DISCUSSION

6.1 VALIDITY OF RESULTS

By using a wire-frame model for the simulation (see 4.2), approximations are applied which deviate the results. The wire-frame simulation with beam elements is compared to an FEM-Model with solid (tetrahedral) elements to show the deviation.

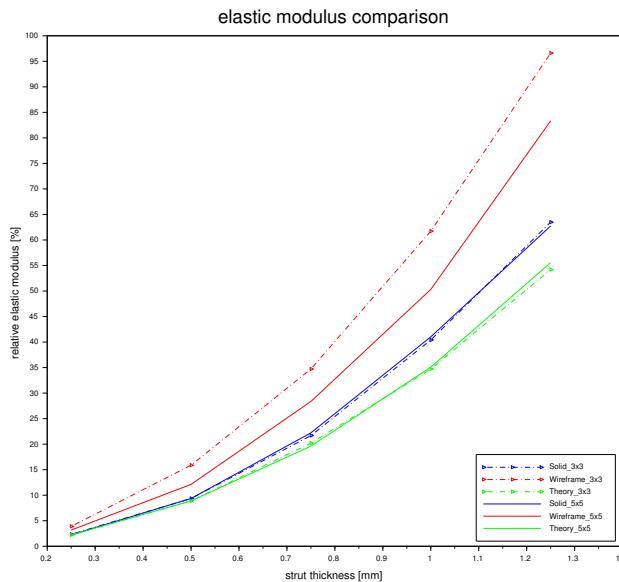


Figure 15.: In this graph the results of different methods to determine the elastic modulus of the meta material with the cubes topology are shown. The dotted lines were evaluated with a scaffold consisting of 3 by 3 by 3 cells. The filled lines show the results with scaffold consisting of 5 by 5 by 5 cells. The green lines show the elastic modulus of only vertical struts within the scaffold with regard of cutting off the border struts according to Figure 16. For the calculation see A.3. The elastic modulus for this theoretical calculation is too low since it does not take the transverse struts into account that would take part of the loading especially for lower porosities (see Figure 17).

In figure 15 it can be seen clearly that the elastic modulus of the scaffold is significantly higher in the wire-frame model than in the solid model. One reason is that the wire-frame model does not take a cutoff of the border struts into account, which allows the scaffold to overhang and use more space than it should. As can be seen in Figure 16, this error is especially high for a small number of cells in a scaffold.

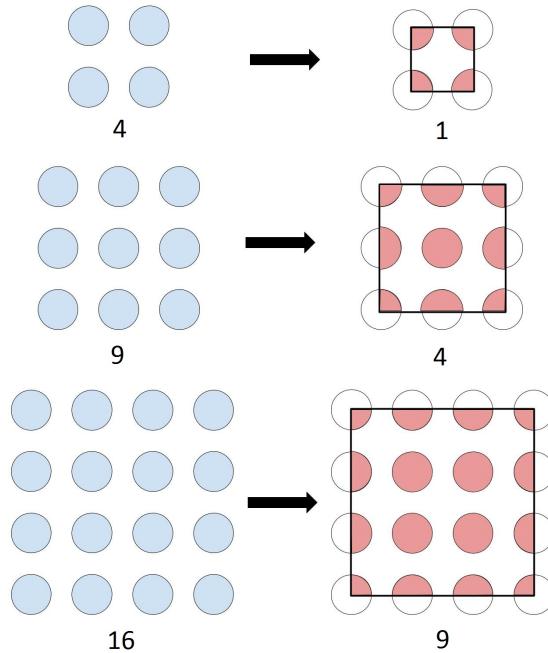


Figure 16.: The stiffness difference of the cubes topology between the cutoff model and the regular model is described by the factor $\frac{i^2}{(i+1)^2}$, where i is the number of cells in the scaffold.

The other reason for a deviation in the results is that beam elements can not fully represent the actual model. It can be seen in Figure 17 that the transverse struts of the solid elements absorb a part of the stress with increasing thickness while the beam elements that are perpendicular to the applied force remain unstressed. Especially for intersections where the angle between two struts is small, the wire-frame approach results in higher stiffnesses because it does not take the intersections between struts into account (see Figure 18). Overall it can be said, that the higher the number of cells and the smaller the strut thicknesses in relation to the cell size, the more accurate is the wire-frame model.

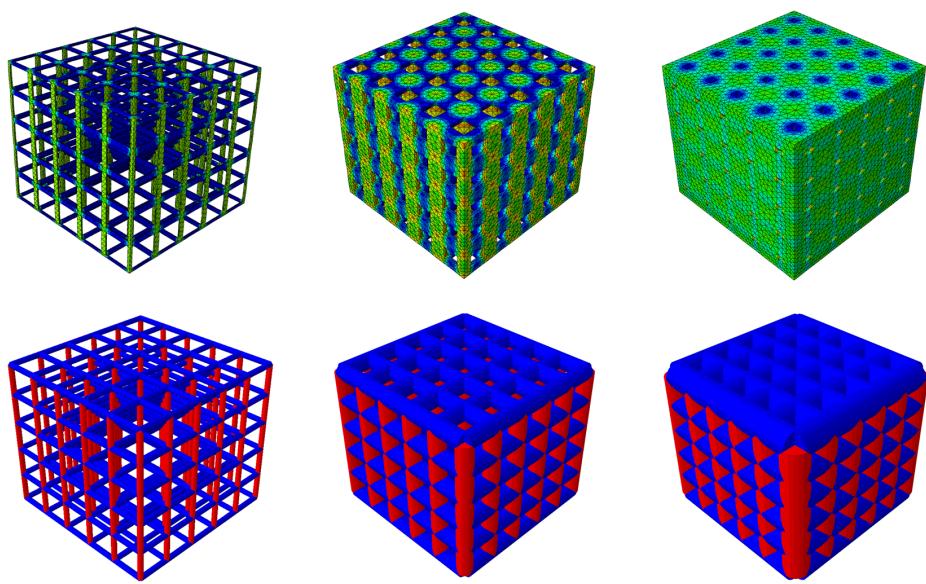


Figure 17.: Comparison of the simulation with solid and with beam elements for different strut thicknesses for the cubes topology. The thicker the struts, the more stress the transverse struts will carry. The wire-frame model does not take this into account.

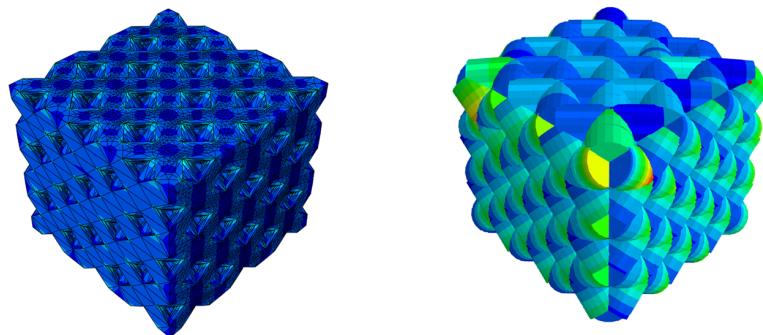


Figure 18.: Comparison of the simulation with solid and with beam elements for the octets topology with low porosity, causing the beam elements to overlap.

6.2 OPTIMIZATION PERFORMANCE

The L-BFGS-B-Algorithm proves to be quite proficient in finding decent results within a reasonable amount of time. The wire-frame model achieves fast results that are within an acceptable error for scaffolds with porosity higher than 60%. Despite the inaccuracy, the wire-frame model does show the correct tendencies for the optimization to step towards descent.

6.3 POSSIBLE FURTHER WORK

6.3.1 *Distortion of cells*

A rather simple method to have further variables to optimize would be to distort any given cell topology into the three main directions.

6.3.2 *Non-cubic cells*

The cell topologies in this thesis are all based on a cubic design to simplify the comparison between them. More complex designs could be evaluated that way. Although it has to be mentioned that if the given optimization goals can be achieved with the current design parameters there is no need for further complexity.

6.3.3 *Combine Cell Topologies*

The overall truss could consist of different topologies to create scaffolds with multiple pore sizes with different Bone-TE-objectives. The bigger pores could offer good interconnectivity to allow nutrient flow while the smaller pores would allow faster bone growth and proliferation.

6.3.4 *Topology Optimization by Bendsøe and Kikuchi*

By allowing struts to be removed once their thickness becomes too small, one could try an optimization similar to the method developed by bentsøe and kikuchi [27].

6.3.5 *Experimental comparison*

The developed scaffolds could be manufactured and tested for their mechanical properties. Additionally, in-vitro and eventually in-vivo experiments would show the actual performance of the scaffolds.

6.3.6 *Other scopes*

Other topics than TE could be investigated further with the presented method. For example, one could use this method on the design of the core of composite sandwich plates.

7

CONCLUSION

In this thesis a method has been introduced to evaluate scaffolds consisting of discrete elements for their mechanical and volumetric properties. Multiple topologies have been introduced and compared for an example goal to fit the mechanical response of the scaffold to that of human cortical bone for bone-TE. As a result several graphs are presented which serve for comparison between the optimization results of the evaluated topologies. The evaluation of the scaffolds by an FEM-simulation with beam elements provides fast results but contains certain errors that need to be addressed for further work. Nevertheless it can be said, that with this method scaffolds can be created that fit certain target values regarding bone-TE.

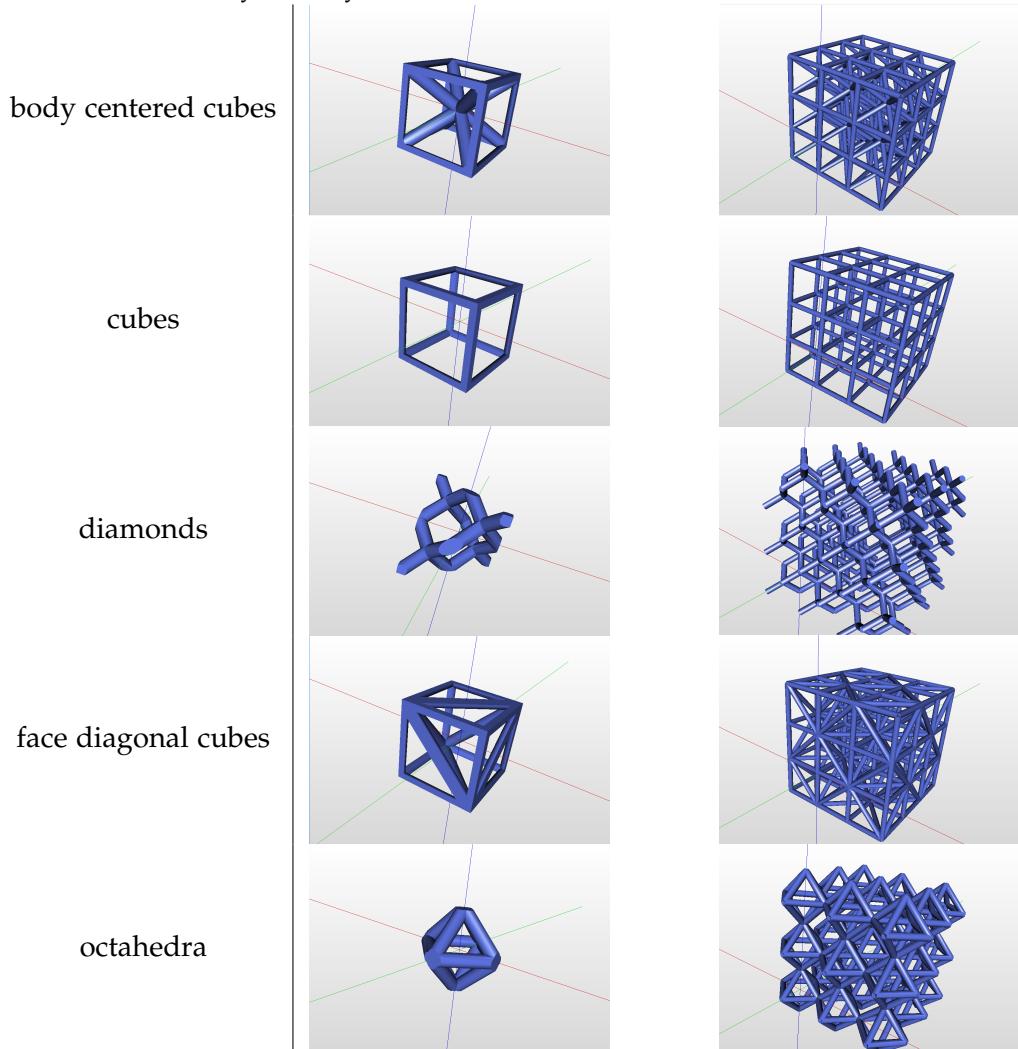
Appendices

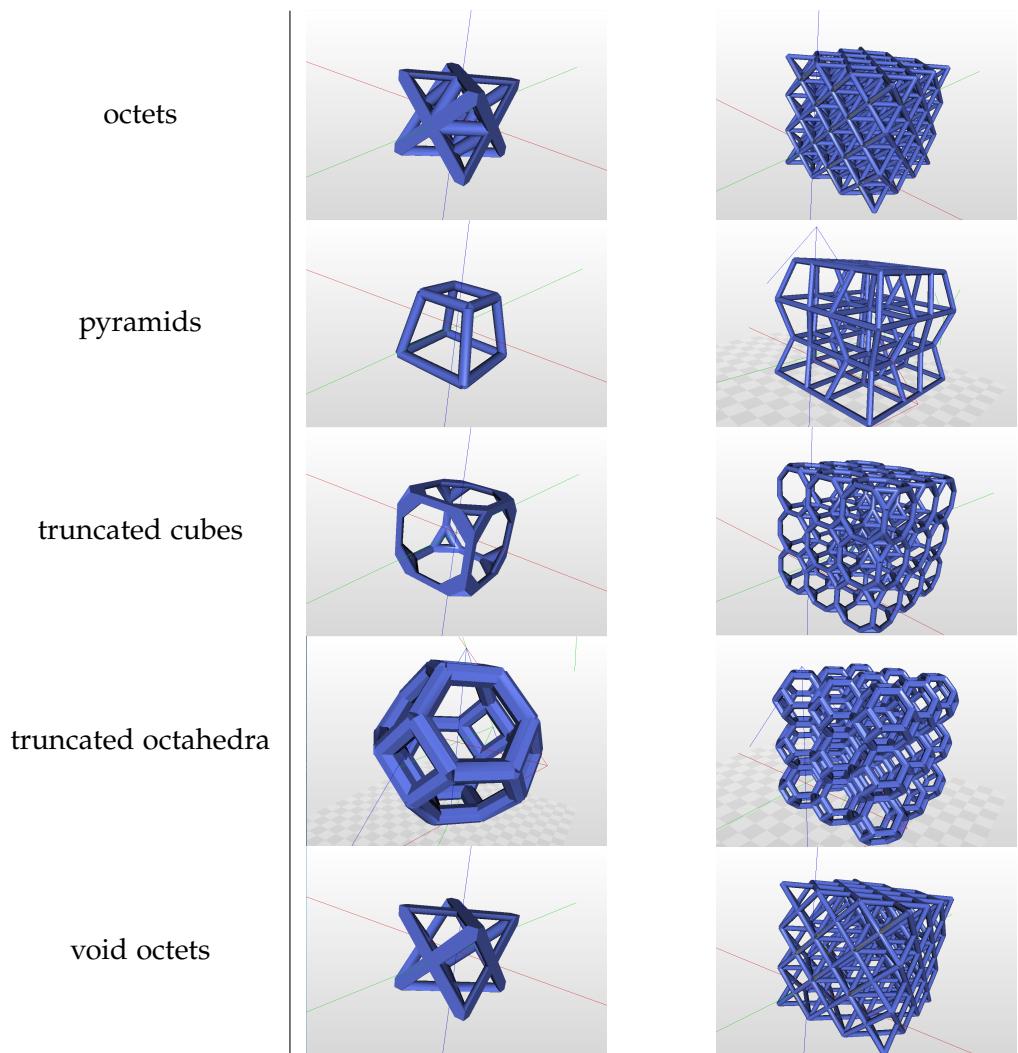
A

APPENDIX

A.1 TOPOLOGIES USED IN THIS THESIS

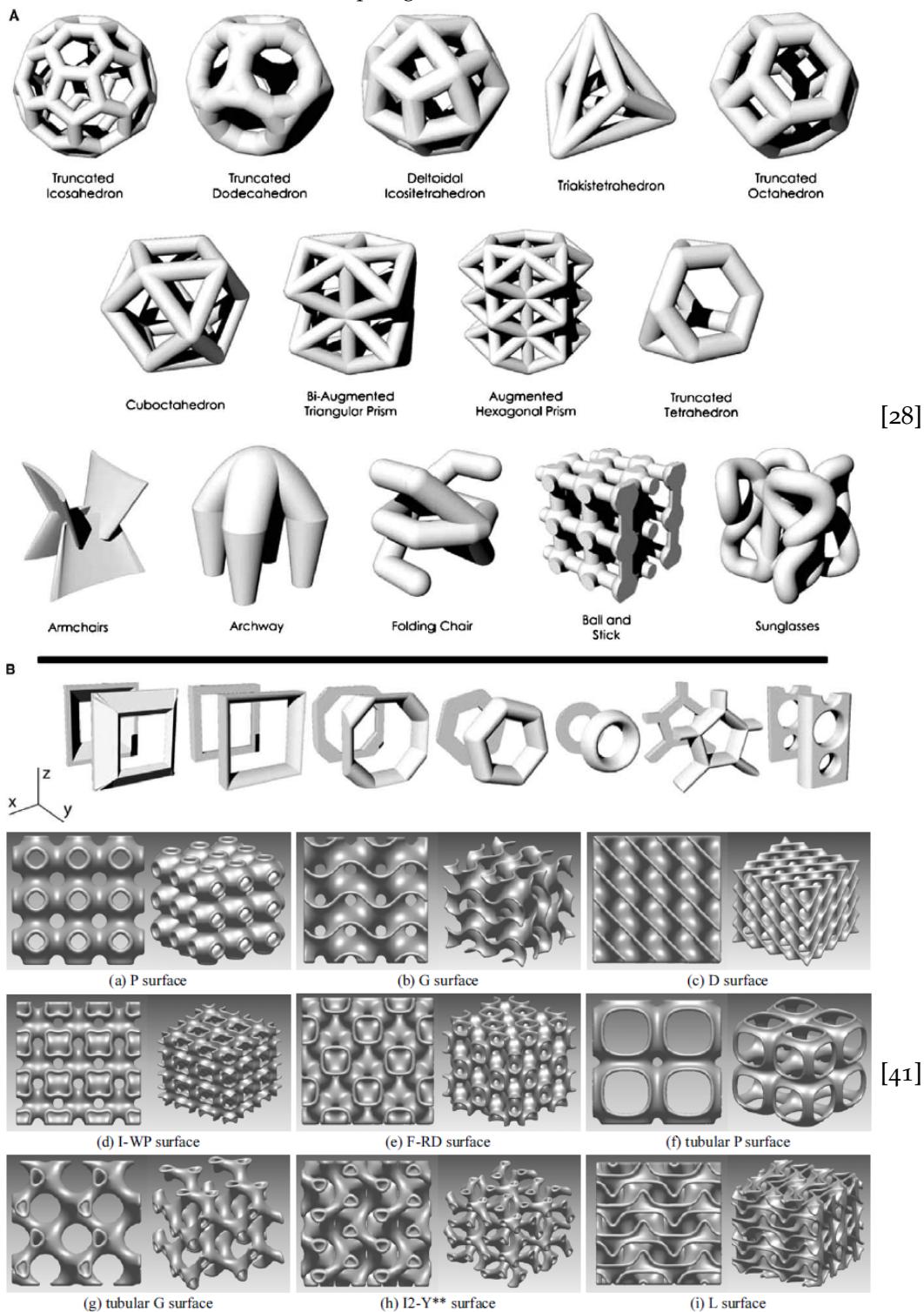
Table 7.: Topologies used in this thesis as a single cell and as a lattice consisting of three by three by three cells.

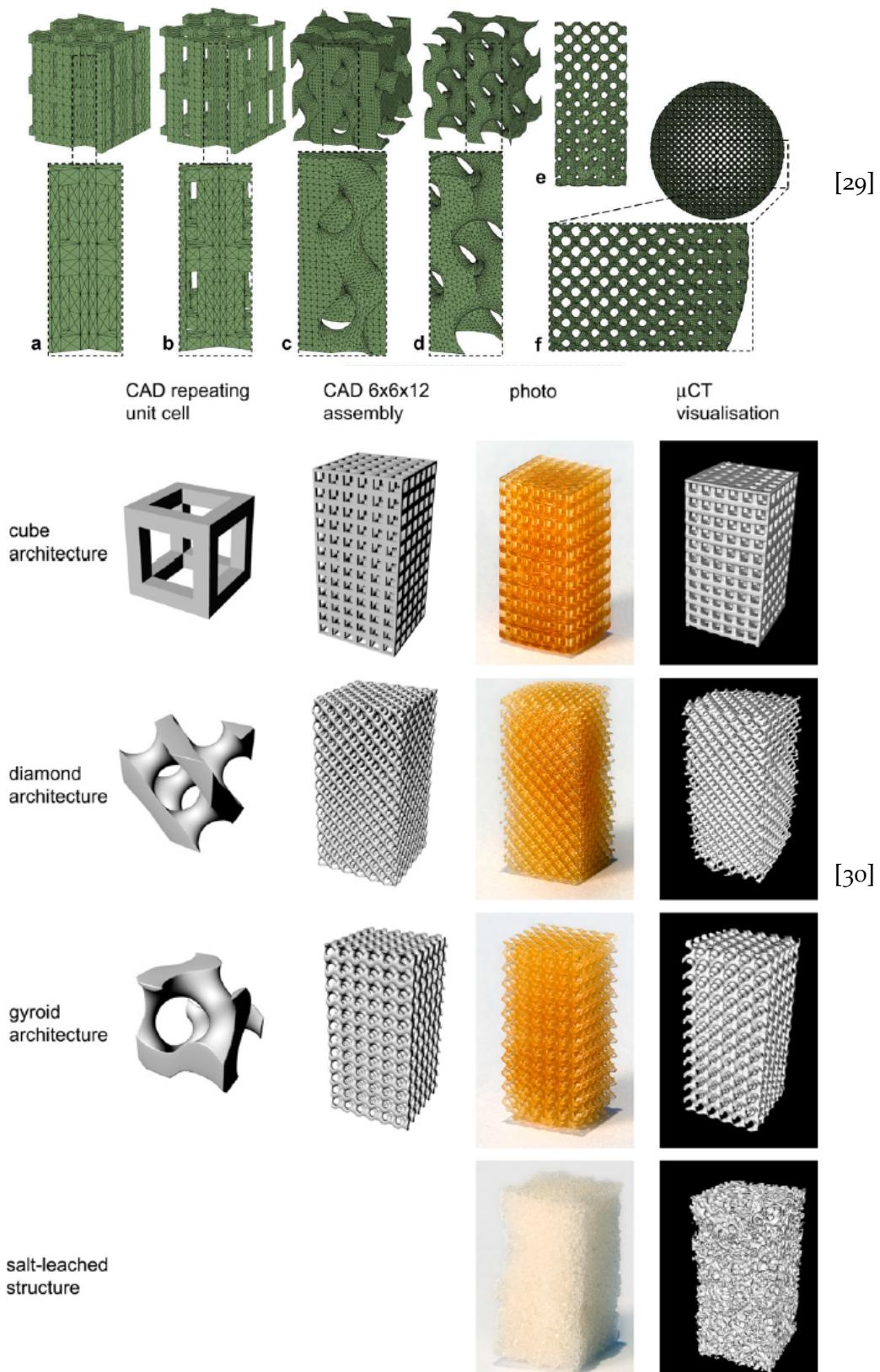


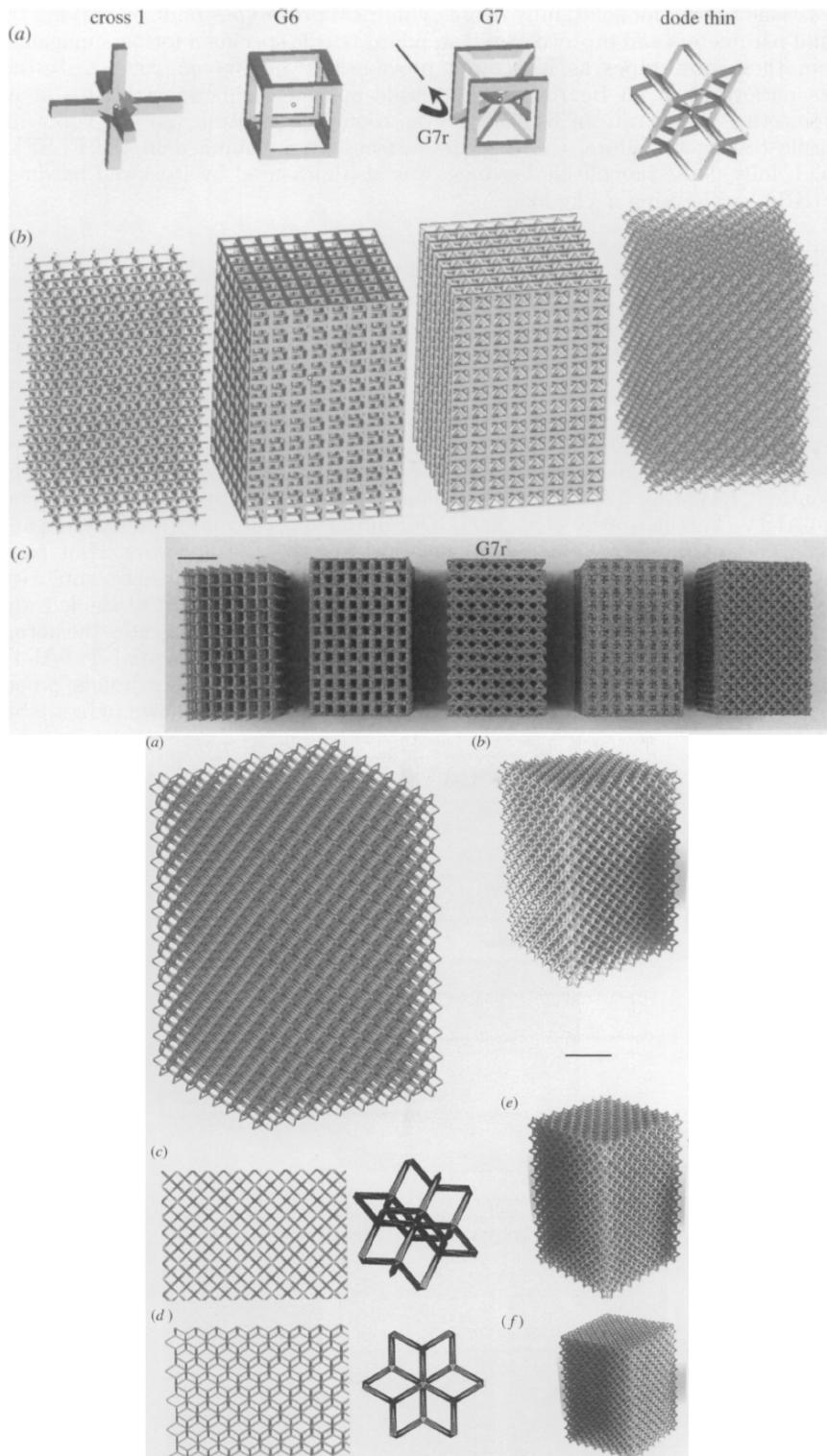


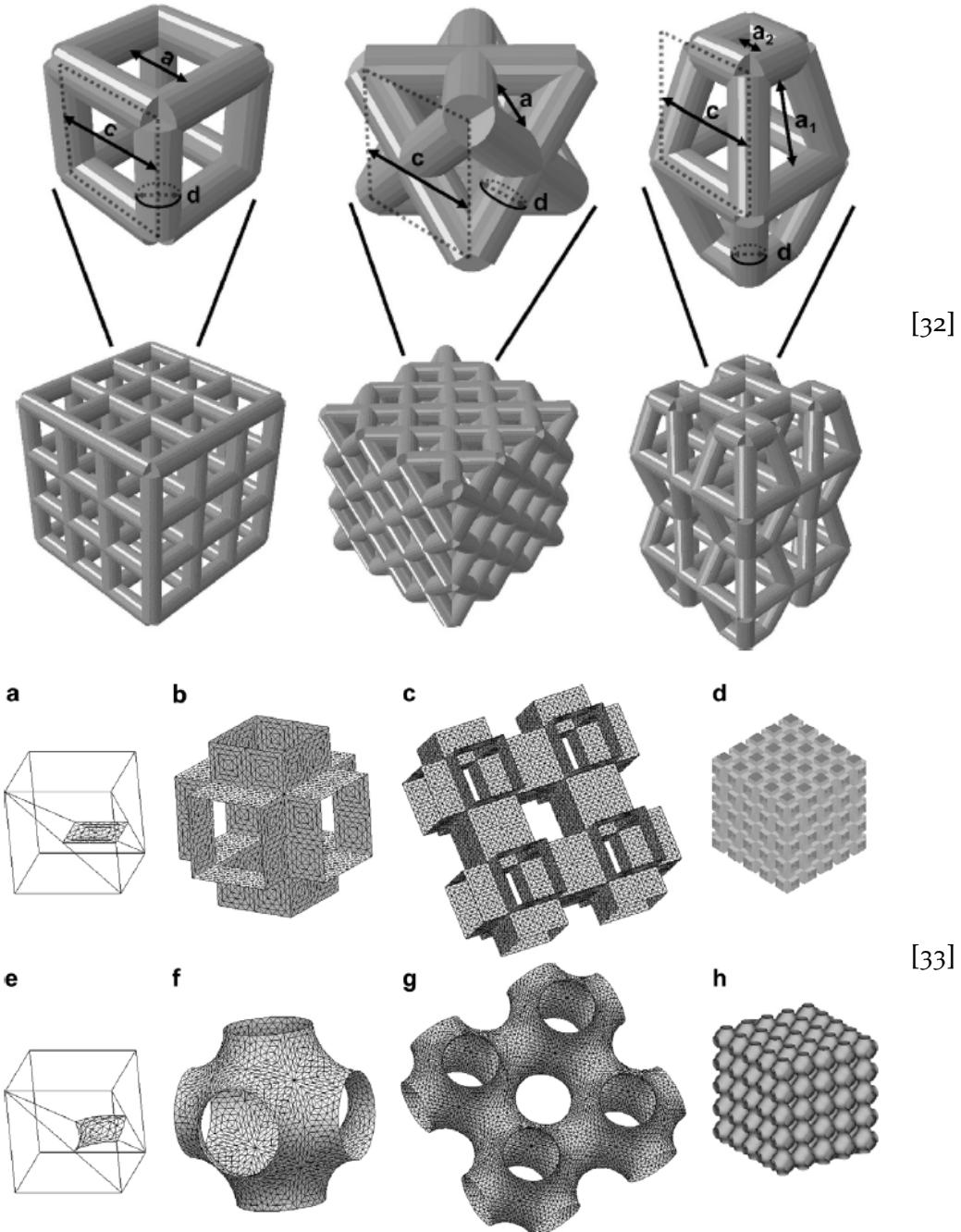
A.2 TOPOLOGIES FROM OTHER RESEARCH

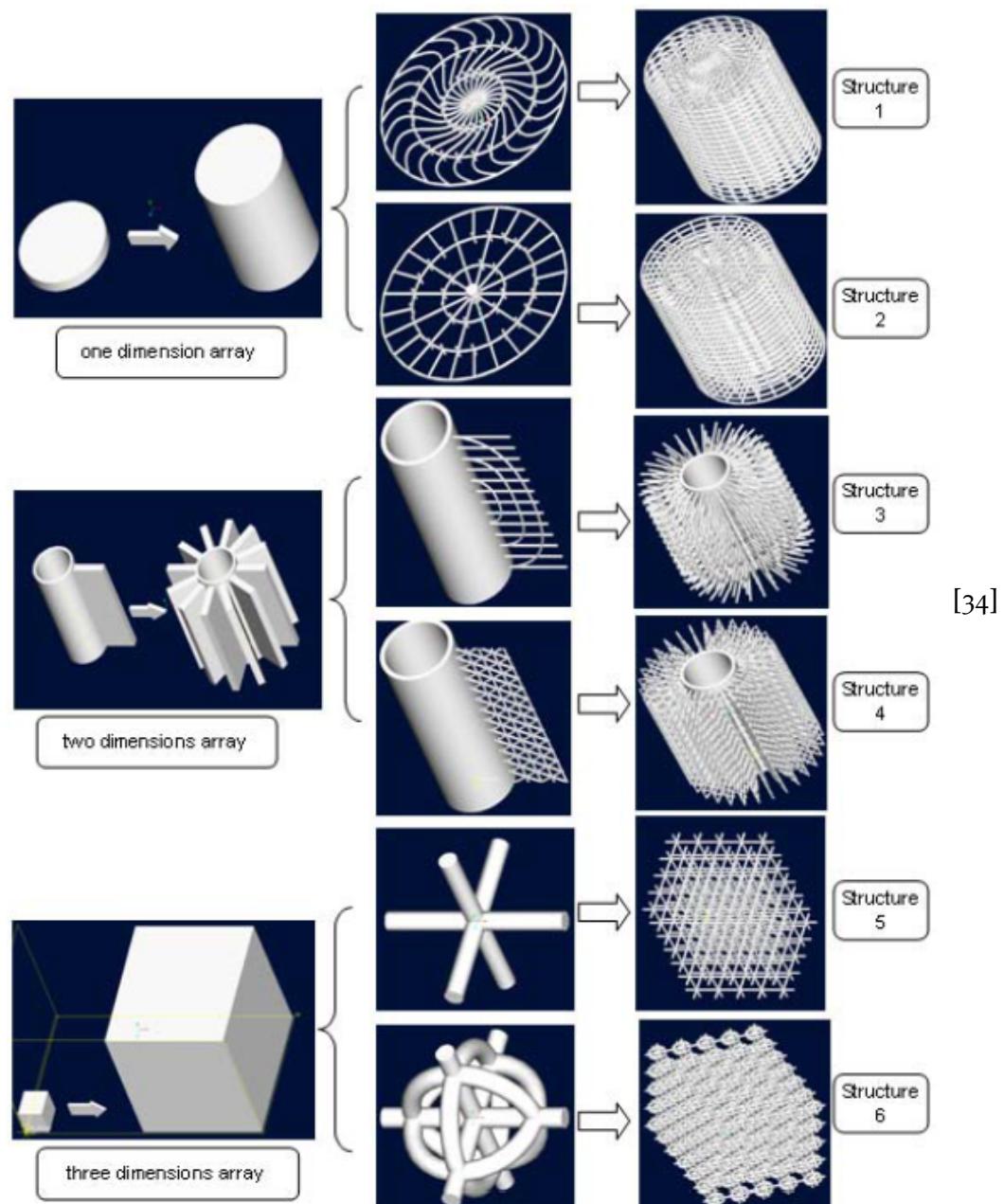
Table 8.: Topologies used in other research.

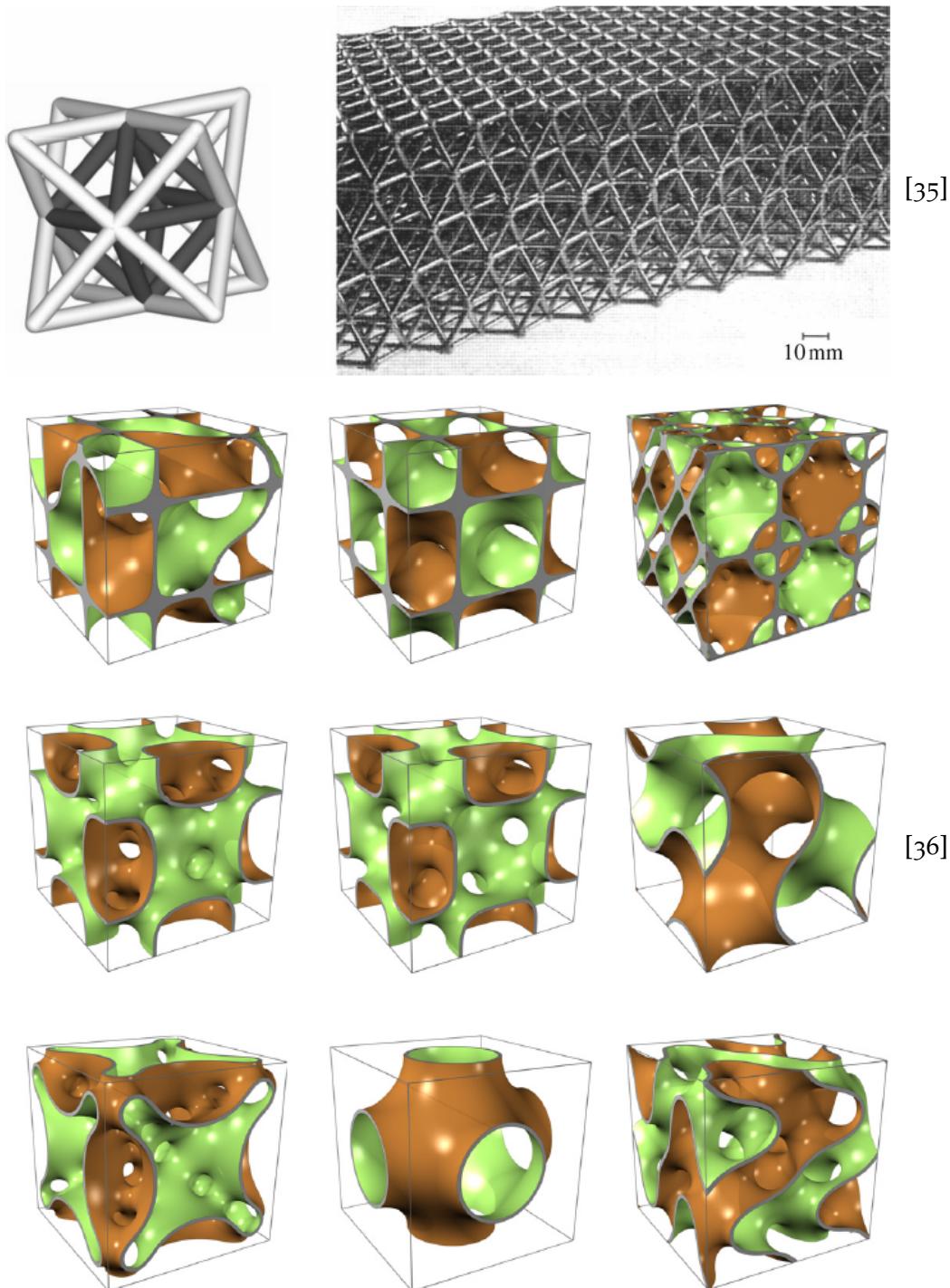


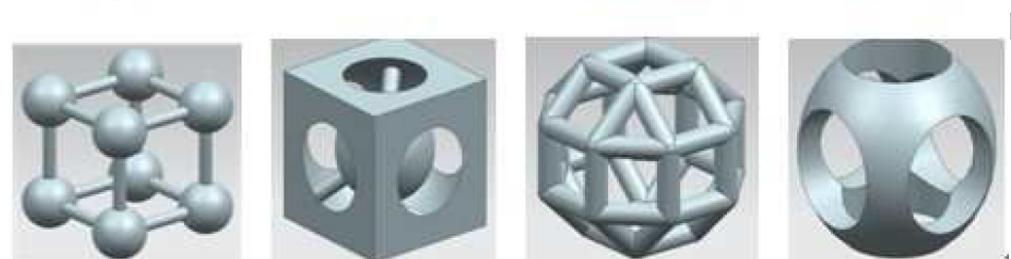
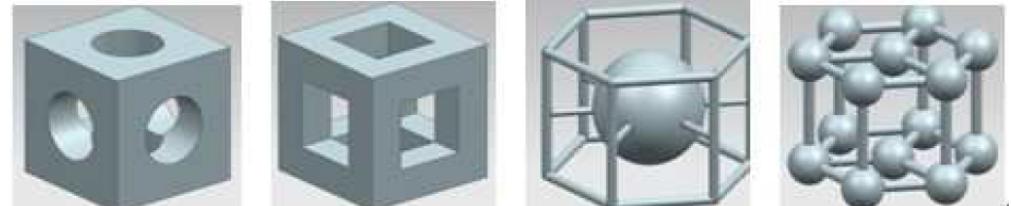
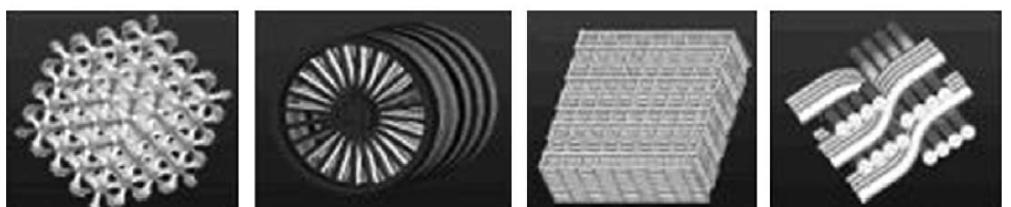
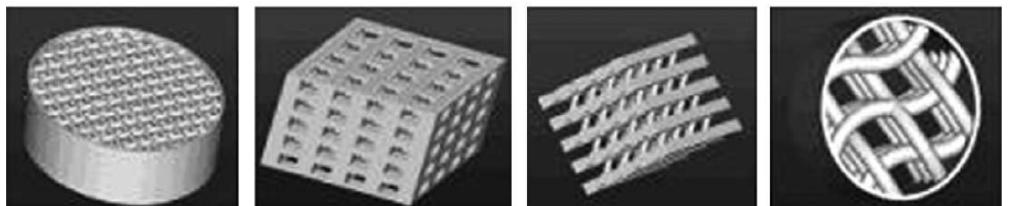
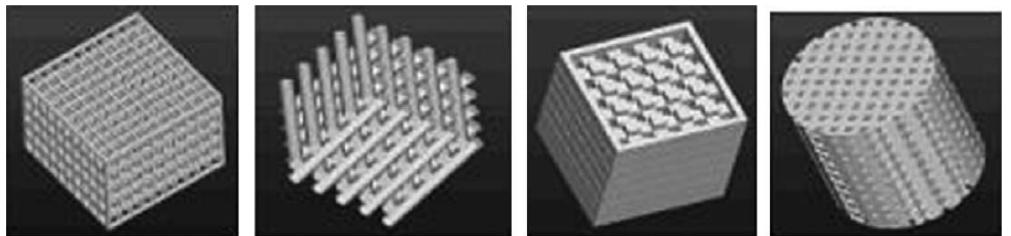












[37]

[38]

A.3 CALCULATION FOR THEORETICAL STIFFNESS OF META-MATERIAL

A.3.1 SciLab Code

```

1 clc
2 clear
3
4 strut_thickness = [0.25:0.25:1.25]
5 displacement_solid = [-3.8d-1 -9.5d-2 -4.1d-2 -2.2d-2 -1.4d-2] // -8.9d-3
6 displacement_wireframe = [-5.7d-2 -1.4d-2 -6.4d-3 -3.6d-3 -2.3d-3]
7 displacement_theory = [-0.10 -0.025 -0.011 -0.0064 -0.0041]
8 displacement_solid = displacement_solid/4
// The force was set at 4N accidental in the simulation.
// Since its all linear elastic this is ok.
9
10 scf()
11 clf()
12 g = gcf()
13 g.figure_position=[200,200]
14 g.figure_size=[1000,1000]
15
16 subplot(221)
17 plot(strut_thickness, -displacement_solid/4.5, 'b->')
18 plot(strut_thickness, -displacement_wireframe/4.5, 'r->')
19 plot(strut_thickness, -displacement_theory/4.5, 'g->')
20 hl=legend(['Solid', 'Wireframe', 'Theory'],1)
21 title("strain 3x3x3", "font_size", 5)
22 xlabel('strut_thickness [mm]', "font_size", 3)
23 ylabel('strain [1]', "font_size", 3)
24
25 modulus_wireframe = -1*4.5./(4.5^2.*displacement_wireframe)
26 modulus_solid = -1*4.5./(4.5^2.*displacement_solid)
27 modulus_theory = -1*4.5./(4.5^2.*displacement_theory)
28
29 subplot(222)
30 plot(strut_thickness, modulus_solid, 'b->')
31 plot(strut_thickness, modulus_wireframe, 'r->')
32 plot(strut_thickness, modulus_theory, 'g->')
33 hl=legend(['Solid', 'Wireframe', 'Theory'],4)
34 title('elastic modulus 3x3x3', "font_size", 5)
35 xlabel('strut thickness [mm]', "font_size", 3)
36 ylabel('relative elastic modulus [%]', "font_size", 3)
37
38
39
40
41
42
43 strut_thickness5 = [0.25:0.25:1.25]
44 displacement_solid5 = [-2.4d-1 -5.7d-2 -2.4d-2 -1.3d-2 -0.85d-2] // -8.9d-3
45 displacement_wireframe5 = [-4.2d-2 -1.1d-2 -4.7d-3 -2.65d-3 -1.6d-3]
46 displacement_theory5 = [-0.06 -0.015 -0.0068 -0.0038 -0.0024]
47 displacement_solid5 = displacement_solid5/4
48
49
50 subplot(223)
51 plot(strut_thickness5, -displacement_solid5/7.5, 'b->')
52 plot(strut_thickness5, -displacement_wireframe5/7.5, 'r->')
53 plot(strut_thickness5, -displacement_theory5/7.5, 'g->')
54 hl=legend(['Solid', 'Wireframe', 'Theory'],1)
55 title("strain 5x5x5", "font_size", 5)
56 xlabel('strut_thickness [mm]', "font_size", 3)
57 ylabel('strain [1]', "font_size", 3)
58
59 modulus_wireframe5 = -1*7.5./(7.5^2.*displacement_wireframe5)
60 modulus_solid5 = -1*7.5./(7.5^2.*displacement_solid5)
61 modulus_theory5 = -1*7.5./(7.5^2.*displacement_theory5)
62 subplot(224)
63 plot(strut_thickness5, modulus_solid5, 'b->')
64 plot(strut_thickness5, modulus_wireframe5, 'r->')
65 plot(strut_thickness5, modulus_theory5, 'g->')
66 hl=legend(['Solid', 'Wireframe', 'Theory'],4)
67 title('elastic modulus 5x5x5', "font_size", 5)
68 xlabel('strut thickness [mm]', "font_size", 3)

```

```
69 | ylabel('relative elastic modulus [%]', "font_size", 3)
70 |
71 | scf(1)
72 | clf()
73 | f = gcf()
74 | f.figure_position=[1300,200]
75 | f.figure_size=[1000,1000]
76 | plot(strut_thickness, modulus_solid, 'b->')
77 | plot(strut_thickness, modulus_wireframe, 'r->')
78 | plot(strut_thickness, modulus_theory, 'g->')
79 | plot(strut_thickness5, modulus_solid5, 'b')
80 | plot(strut_thickness5, modulus_wireframe5, 'r')
81 | plot(strut_thickness5, modulus_theory5, 'g')
82 | hl=legend(['Solid_3x3', 'Wireframe_3x3', 'Theory_3x3', 'Solid_5x5',
83 | 'Wireframe_5x5', 'Theory_5x5'],4)
84 | title('elastic modulus comparison', "font_size", 5)
85 | xlabel('strut thickness [mm]', "font_size", 3)
86 | ylabel('relative elastic modulus [%]', "font_size", 3)
```

A.4 CODE

A.4.1 *truss_builder.py*

```

69 # Contains the number of different ratios on each cell (for topology optimization):
70 truss_ratio_library = dict(cubes=0,
71                           inv_cubes=0,
72                           body_centered_cubes=0,
73                           truncated_cubes=1,
74                           face_diagonal_cubes=0,
75                           face_diagonal_cubes_alt=0,
76                           octetrahedrons=0,
77                           octahedrons=0,
78                           void_octetrahedrons=0,
79                           diamonds=0,
80                           templar_crosses=3,
81                           templar_alt_crosses=3,
82                           templar_alt2_crosses=3,
83                           pyramids=1,
84                           file_super_truss=0,
85                           tetroctas=0,
86                           truncated_octahedrons=1)
87 # Dictionary of materials: E_Modulus: [N/mm^2] = [MPa]
88 material_library = dict(MED610=dict(name='MED610', E_Modulus=2.5e3, poisson_ratio=0.33), # Med-grade Polymer
89                         PLA=dict(name='PLA', E_Modulus=2.5e3, poisson_ratio=0.33), # Poly-lactic acid
90                         TCP=dict(name='TCP', E_Modulus=22e3, poisson_ratio=0.33), # Tricalciumphosphate
91                         HA=dict(name='HA', E_Modulus=6e3, poisson_ratio=0.33), # Hydroxyapatite
92                         Titanium=dict(name='Titanium', E_Modulus=116e3, poisson_ratio=0.32),
93                         Relative=dict(name='Relative', E_Modulus=100, poisson_ratio=0.33)
94                     )
95 #####
96 # INPUTS
97 # Directory where abaqus loads and saves files. The directory will be created if it doesn't exist already.
98 # This generated data is rather large:
99 # This generated data is very small:
100 inputs['calculating_directory'] = "C://abaqus_temp/"
101 # Directory where outputs such as csv and pickle get saved. The directory will be created if it doesn't exist already.
102 # Affix to every file in the calculating directory:
103 inputs['output_directory'] = 'C://Users/maxe/Dropbox/Master_Thesis/outputs/final/superfinal/hypothesis/'
104 inputs['job_name'] = "temp_output"
105 # Name (and therefore topology) of the truss. Multiple names can be put in to iterate through all.
106 # For a single input, put it in brackets. For example: ['cubes']
107 # inputs['truss_names'] = ['body_centered_cubes'] # , 'diamonds', 'truncated_cubes']
108 inputs['truss_names'] = ['cubes', 'octetrahedrons']
109 # inputs['truss_names'] = ['cubes', 'body_centered_cubes', 'truncated_cubes', 'face_diagonal_cubes_alt',
110 #                           'octetrahedrons', 'octahedrons', 'void_octetrahedrons', 'diamonds', 'pyramids',
111 #                           'truncated_octahedrons']
112 #
113 #
114 # Material. See material_library to add or see materials.
115 inputs['material'] = material_library['Relative']
116
117 # Cell topology
118 # Number of cells in one direction (Total number of cells is number_of_cells ** 2):
119 inputs['number_of_cells'] = 5
120 # Length of one cell (Total size is cell_size * number_of_cells):
121 inputs['cell_size'] = 1 # [mm]
122 # Minimal Thickness that can be used:
123 inputs['strut_min_thickness'] = 0.1 # [mm]
124 # A list will be created inputs['strut_thickness_multipliator'] that has the needed length depending on cell topology
125 # This lists elements will all be inputs['strut_min_thickness'] * inputs['strut_thickness_multipliator']
126 inputs['strut_thickness_multipliator'] = 1
127 inputs['cell_ratio_multiplier'] = 0.5
128 #####
129 #####
130 #####
131 # OPTIONS
132
133 # Creates the loading steps for the wireframe evaluation:
134 options['create_steps'] = True
135 # Submits the job and evaluates the created steps:
136 options['submit_job'] = True
137 # Generates the solid model of the truss and exports it into an stl-file. Needed to calculate volumetric outputs:
138 options['stl_generate'] = True
139 # Cuts the borders of the truss off:
140 options['cutoff'] = True
141 # Runs the program with the GUI (Generated User Interface). This blocks submitting of the job:
142 options['gui'] = False
143 # Overwrites the csv result_file instead of appending the new results

```

```

144 # The result gets saved in the output_directory
145 # If there hasn't been an output file yet, this needs to be True:
146 options['overwrite_csv'] = True
147 # Overwrites the pickled result file instead of appending the new results
148 # (The result gets saved in the output_directory)
149 # If there hasn't been an output file yet, this needs to be True:
150 options['overwrite_pickle'] = True
151
152 # Viewers
153 # Opens the stl (STereoLithography) after completion:
154 options['stl_view'] = False
155 # Opens the odb (Output DataBase) after completion:
156 options['odb_view'] = False
157 # Opens the csv (Comma Separated Values) of the output after completion:
158 options['csv_view'] = False
159
160 # Cross Section Input
161 # Cross Section of the Struts for the Solid and Stl Model. Can be "square", "hexagon", "octagon" or "dodecagon":
162 options['strut_cross_section'] = 'dodecagon'
163
164 # Version Input
165 # To determine Version: use windows command prompt: "abaqus cae nogui" and then ">>> version" or ">>> print(version)"
166 # If the student version is used add SE in the end. F.e. '6.14-2SE'
167 # This is used for defining the path of where to find the abaqus_plugin stlExport. See class Script : __init__()
168 options['abaqus_version'] = '6.14-1'
169
170 # File Path
171 # This is usually C:/SIMULIA/Abaqus/
172 options['abaqus_path'] = "C:/Program Files/Abaqus/"
173 ##### METHOD to run the engine. Possible entries are 'single_run', 'loop' or 'optimization'
174 options['method'] = 'loop'
175
176 # LOOP specific (applies for options['method'] == 'loop')
177 # Loop over first variable:
178 options['loop1_variable'] = 'number_of_cells'
179 # List of all the values to evaluate
180 options['loop1_values'] = [a for a in range(1, 6)]
181
182
183 # Optional loop over second variable:
184 # Use 'None' if there is to be no loop
185 options['loop2_variable'] = 'None'
186 # Use [0] if there is to be no loop
187 options['loop2_values'] = [0]
188
189 # OPTIMIZATION specific
190 # Decides what to optimize_for:
191 options['optimization_variables'] = 'strut_thickness_multiplicator'
192 # Bounds of the optimization variables. This only applies to some algorithms.
193 options['bounds'] = (1, 20)
194 # Define Fitness variables, their target value, their norming and their weighting.
195 # options['fitness_variables'] = {'Sigma_z': [30e3, 1e-2, 1]}
196 options['fitness_variables'] = {'Sigma_x': [12e3, 1e-2, 1],
197 'Sigma_y': [12e3, 1e-2, 1],
198 'Sigma_z': [15e3, 1e-2, 1]}
199
200 # Plot the fitness while optimizing:
201 options['plot_fitness'] = True
202 # Defines the algorithm used for the optimization:
203 options['algorithm'] = 'L-BFGS-B'
204 # Possible algorithms:
205 # 'Nelder-Mead', 'Powell', 'CG', 'BFGS', 'Newton-CG', 'L-BFGS-B', 'TNC', 'COBYLA', 'SLSQP', 'dogleg', 'trust-ncg'
206 # Options for the optimization. See scipy optimization toolbox for further info:
207 options['options'] = {'disp': True, 'eps': 0.2, 'ftol': 0.2}
208 # Reruns the results with generating stl and writing all results into one csv:
209 options['run_results'] = True
210 ##### OUTPUT
211
212 # Decide what to save in the csv
213 options['output'] = dict()
214 # General:
215 options['output']['Step'] = True
216 options['output']['Truss_Name'] = True
217 options['output']['Fitness'] = True
218 # Geometric properties:

```

```

219 options['output']['Cell_size'] = True
220 options['output']['Strut_Thickness'] = True # =Strut_Thicknesses[0]
221 options['output']['Pore_size'] = True
222 # Mechanic properties:
223 options['output'][ "Young's Modulus"] = True
224 options['output'][ 'Shearing Modulus'] = True
225 options['output'][ "Poisson's Ratio"] = True
226 # Volumetric properties: (will only be calculated if the STL is generated)
227 options['output'][ 'Volume'] = True
228 options['output'][ 'Porosity'] = True
229 options['output'][ 'Void_ratio'] = True
230 options['output'][ 'Surface_Area'] = True
231 # Optimization properties:
232 options['output'][ 'X'] = True # Strut_Thickness_Multiplicator
233
234 ##### NO MORE INPUT #####
235 # ##### NO MORE INPUT #####
236 # ##### NO MORE INPUT #####
237 ##### NO MORE INPUT #####
238 ##### NO MORE INPUT #####
239 if options['gui']:
240     options['submit_job'] = False
241     print("Job will not be submitted if the Script is run with the GUI.")
242 if not options['stl_generate']:
243     options['stl_view'] = False
244     print("Stl cannot be viewed if it is not generated")
245 if not options['submit_job']:
246     options['odb_view'] = False
247     print("Odb cannot be viewed if the job is not submitted")
248
249 options['read_output'] = options['submit_job']
250
251 # Create folders if they don't exist yet.
252 if not os.path.exists(inputs['calculating_directory']):
253     os.mkdir(inputs['calculating_directory'])
254
255 if not os.path.exists(inputs['output_directory']):
256     os.mkdir(inputs['output_directory'])
257
258 if options['method'] == 'optimization':
259     # Initialize csv
260     file = open(inputs['output_directory'] + "optimization_results.csv", 'w')
261     file.write("x, fun, nit, success, nfev\n")
262     file.close()
263     # Initialize pickle file
264     pickle.dump(list(), open(inputs['output_directory'] + "pickled_results", 'wb'))
265
266 # Loop over the different truss topologies:
267 for name in inputs['truss_names']:
268     try:
269         inputs['truss_name'] = name
270
271         bounds[options['optimization_variables']] = [options['bounds']]
272
273         inputs['strut_thickness_multipliator'] = list()
274         # Multiplied by strut_min_thickness results in the actual thickness of the struts
275         bounds['strut_thickness_multipliator'] = list()
276         # Multiplied by strut_min_thickness results in the actual thickness of the struts bounds
277         for i in range(0, truss_thicknesses_library[name]):
278             inputs['strut_thickness_multipliator'].append(inputs['strut_thickness_multiplier'])
279             bounds['strut_thickness_multipliator'].append(options['bounds'])
280         # Best thicknesses for titanium (1.5mm cell_size)(0.1mm strut_min_thickness) (3x3x3)
281         # inputs['strut_thickness_multipliator'] = [4.71177218, 4.14965115, 4.80241845]
282
283         inputs['cell_ratio'] = list()
284         bounds['cell_ratio'] = list()
285         for j in range(0, truss_ratio_library[name]):
286             inputs['cell_ratio'].append(inputs['cell_ratio_multiplier'])
287             bounds['cell_ratio'].append(options['bounds'])
288         # Best ratio for templar_alt2_crosses:
289         # inputs['cell_ratio'] = [0.3, 0.8, 0.2]
290         # printable:
291         # inputs['cell_ratio'] = [0.5, 0.8, 0.35]
292
293         inputs['output_file'] = inputs['output_directory'] + 'optim_output_' + str(name)

```

```

294
295     if options['overwrite_csv']:
296         result_file = open(str(inputs['output_file']) + '.csv', 'w')
297     if options['output']['Step']:
298         result_file.write('Step, ')
299     if options['output']['Truss_Name']:
300         result_file.write('Truss Name, ')
301     if options['output']['Fitness']:
302         result_file.write('Fitness, ')
303     if options['output']['Cell_size']:
304         result_file.write('Cell Size [mm], ')
305     if options['output']['Strut_Thickness']:
306         result_file.write('Strut Thickness [mu-m], ')
307     if options['output']['Pore_size']:
308         result_file.write('Pore Size 1[mu-m], Pore Size 2[mu-m], Pore Size 3[mu-m], Pore Size 4[mu-m], ')
309     if options['output'][ "Young's Modulus"]:
310         result_file.write('Sigma_z [MPa], Sigma_y [MPa], Sigma_x [MPa], ')
311     if options['output'][ 'Shearing Modulus']:
312         result_file.write('Tau_yz [MPa], Tau_xz [MPa], Tau_xy [MPa], ')
313     if options['output'][ "Poisson's Ratio"]:
314         result_file.write('v21 [1], v31 [1], v32 [1], ')
315     if options['output'][ 'Volume']:
316         result_file.write('Volume [mm^3], ')
317     if options['output'][ 'Porosity']:
318         result_file.write('Porosity [%], ')
319     if options['output'][ 'Void_ratio']:
320         result_file.write('Void_ratio [1], ')
321     if options['output'][ 'Surface_Area']:
322         result_file.write('Surface_Area [mm^2], ')
323     if options['output'][ 'X']:
324         result_file.write('Strut_Thickness_Multiplicator [1], ')
325     result_file.write('\n')
326     result_file.close()
327
328 if options['run_results']:
329     result_file2 = open(inputs['output_directory'] + "best_results" + '.csv', 'w')
330     if options['output']['Step']:
331         result_file2.write('Step, ')
332     if options['output']['Truss_Name']:
333         result_file2.write('Truss Name, ')
334     if options['output']['Fitness']:
335         result_file2.write('Fitness, ')
336     if options['output']['Cell_size']:
337         result_file2.write('Cell Size [mm], ')
338     if options['output']['Strut_Thickness']:
339         result_file2.write('Strut Thickness [mu-m], ')
340     if options['output']['Pore_size']:
341         result_file2.write('Pore Size 1[mu-m], Pore Size 2[mu-m], Pore Size 3[mu-m], Pore Size 4[mu-m], ')
342     if options['output'][ "Young's Modulus"]:
343         result_file2.write('Sigma_z [MPa], Sigma_y [MPa], Sigma_x [MPa], ')
344     if options['output'][ 'Shearing Modulus']:
345         result_file2.write('Tau_yz [MPa], Tau_xz [MPa], Tau_xy [MPa], ')
346     if options['output'][ "Poisson's Ratio"]:
347         result_file2.write('v21 [1], v31 [1], v32 [1], ')
348     if options['output'][ 'Volume']:
349         result_file2.write('Volume [mm^3], ')
350     if options['output'][ 'Porosity']:
351         result_file2.write('Porosity [%], ')
352     if options['output'][ 'Void_ratio']:
353         result_file2.write('Void_ratio [1], ')
354     if options['output'][ 'Surface_Area']:
355         result_file2.write('Surface_Area [mm^2], ')
356     if options['output'][ 'X']:
357         result_file2.write('Strut_Thickness_Multiplicator [1], ')
358     result_file2.write('\n')
359     result_file2.close()
360
361 if options['overwrite_pickle']:
362     output_old = list()
363     file = open(inputs['output_file'] + '_pickle', 'wb')
364     pickle.dump(output_old, file)
365     file.close()
366
367 # Counts the number of function evaluations. Careful, this is global
368 universal_counter = 0

```

```

369 #####
370 # Calling the engine, depending on the chosen method
371 if options['method'] == 'single_run':
372     options['plot_fitness'] = False
373     objective_function(inputs=inputs, options=options)
374
375 elif options['method'] == 'loop':
376
377     options['plot_fitness'] = False
378     for loop1 in options['loop1_values']:
379         for loop2 in options['loop2_values']:
380             inputs[options['loop1_variable']] = loop1
381             inputs[options['loop2_variable']] = loop2
382             # SPECIAL RUN
383             inputs['cell_size'] = 5 / inputs['number_of_cells']
384             inputs['strut_min_thickness'] = inputs['cell_size'] / 5
385             # END SPECIAL RUN
386             objective_function(inputs=inputs, options=options)
387
388 elif options['method'] == 'optimization':
389     print("Optimization Input:")
390     print("xo= " + str(inputs[options['optimization_variables']]))
391     print("inputs= " + str(inputs) + ", " + str(options))
392     print("options= " + str(options))
393     print("method= " + str(options['algorithm']))
394     print("bounds= " + str(bounds[options['optimization_variables']]))
395     print("options= " + str(options['options']))
396     print("#####\n\n")
397
398     result = optimize.minimize(optimizer,
399                               x0=inputs[options['optimization_variables']],
400                               args=(inputs,
401                                     options),
402                               method=options['algorithm'],
403                               bounds=bounds[options['optimization_variables']],
404                               options=options['options'])
405
406     # Save result as CSV:
407     file = open(inputs['output_directory'] + "optimization_results.csv", 'a')
408     key_list = ['x', 'fun', 'nit', 'success', 'nfev']
409     for key in key_list:
410         file.write(str(result[key]) + ", ")
411     file.write("\n")
412     file.close()
413
414     # Save result as pickled file:
415     result_pickle = {'inputs': inputs, 'options': options,
416                      'optimization': {'x': result['x'], 'nfev': result['nfev'],
417                                      'nit': result['nit'], 'success': result['success']}}
418     print(result_pickle['optimization'])
419     output_old = pickle.load(open(inputs['output_directory'] + "pickled_results", 'rb'))
420     output_old.append(result_pickle)
421     file = open(inputs['output_directory'] + "pickled_results", 'wb')
422     pickle.dump(output_old, file)
423     file.close()
424
425     print("#####\n\n")
426 else:
427     print("options['method'] does not contain a valid keyword. possible entries are: "
428          "'single_run', 'loop' or 'optimization'.")
429 #####
430
431 if options['csv_view']:
432     open_csv(str(inputs['output_file']) + '.csv')
433 except FileNotFoundError:
434     print("Abaqus faced an error.\nContinuing with the other topologies.")
435
436 # Run the results with calculating volumetric properties
437 if options['method'] == 'optimization' and options['run_results']:
438     output = pickle.load(open(inputs['output_directory'] + "pickled_results", 'rb'))
439     print("Output: " + str(output))
440     for i in range(0, len(output)):
441         inputs = output[i]['inputs']
442         options = output[i]['options']
443         inputs[options['optimization_variables']] = output[i]['optimization']['x']

```

```
444     # inputs['number_of_cells'] = 3
445     inputs['output_file'] = inputs['output_directory'] + "best_results"
446     options['stl_generate'] = True
447     options['cutoff'] = True
448
449     # Initialize best_results_pickle
450     file = open(inputs['output_file'] + "_pickle", 'wb')
451     pickle.dump(list(), file)
452     file.close()
453
454     objective_function(inputs=inputs, options=options)
```

A.4.2 evaluation.py

```

1 """
2 evaluation.py
3 This file contains the main steps of the program such as initiation of optimization values, initiation of
4 optimization and generation of the stl for the final solution.
5 Each solution is an object with the following properties:
6 Nodal points, start and finish of every strut, strut thickness of each strut, pore size, porosity, surface area
7 Each solution might have the following properties:
8 Material constants, Cell types (which are repeated through the truss), cell size, thickness of each strut of a cell
9 """
10 import os
11 import subprocess
12 import time
13 import pickle
14 from library_truss import generate_truss
15 from Class_Script import Script
16 import statistics
17 from matplotlib import pyplot
18 import warnings
19 import matplotlib.cbook
20 import numpy
21
22 warnings.filterwarnings("ignore", category=matplotlib.cbook.mplDeprecation)
23
24 #####
25 # HEADER
26
27
28
29 # RUN THE ABAQUS SCRIPT VIA COMMAND LINE
30 def run(script, gui):
31     print("Running Abaqus Script")
32     t0 = time.time()
33     os.chdir(str(script.filename[0]))
34
35     if gui:
36         subprocess.run('abaqus cae script=' + ''.join(script.filename), shell=True)
37     else:
38         subprocess.run('abaqus cae noGUI=' + ''.join(script.filename), shell=True)
39     t1 = time.time()
40     print("Time elapsed: " + str(t1 - t0) + " seconds\n")
41
42
43 # OPENS THE ABAQUS ODB VIEWER VIA COMMAND LINE
44 def odb_viewer(script):
45     print("Starting Abaqus Viewer")
46     subprocess.Popen('abaqus viewer database=' + script.filename[0] + script.filename[1], shell=True)
47
48
49 # OPENS THE STL FILE WITH SELECTED DEFAULT PROGRAM VIA COMMAND LINE
50 def stl_viewer(script):
51     print("Starting STL Viewer")
52     subprocess.Popen(script.filename[0] + script.filename[1] + ".stl", shell=True)
53
54
55 # READS IN THE DISPLACEMENTS OF THE SIDES OF THE CUBES FOR EACH STEP/LOADING CONDITION AND CALCULATES COMPLIANCE FROM IT
56 def read_results(script, x):
57     def read_stress(result, name, plane, direction):
58         list_of_coordinates = list()
59         for point in result[name][plane]:
60             list_of_coordinates.append(float(point[direction]))
61         displacement = statistics.median(list_of_coordinates)
62         # print("Average displacement " + name + ": " + str(round(displacement * 1e3, 3)) + " µm")
63         young = abs(applied_force / ((script.truss.cell_size * script.truss.number_of_cells +
64                                         script.truss.cells[0].strut_thickesses[0]) * displacement))
65         # print("Elastic Modulus in " + name + " step into " + plane +
66         #       " direction: " + str(round(young / 1e6, 3)) + " MPa")
67         return young
68
69     def read_shearing(result, name, plane, direction):
70         list_of_coordinates = list()
71         for point in result[name][plane]:

```

```

72         list_of_coordinates.append(float(point[direction]))
73         displacement = statistics.median(list_of_coordinates)
74         # print("Average displacement " + name + ": " + str(round(displacement * 1e6, 3)) + " μm")
75         shear = abs(applied_force / ((script.truss.cell_size * script.truss.number_of_cells +
76                                         script.truss.cells[0].strut_thicknesses[0]) * displacement))
77         return shear
78
79     def read_displacement(result, step, plane, direction):
80         list_of_coordinates = list()
81         for point in result[step][plane]:
82             list_of_coordinates.append(float(point[direction]))
83         displacement = statistics.median(list_of_coordinates)
84         # print("Average displacement " + step + ": " + str(round(displacement * 1e6, 6)) + " μm")
85         return displacement
86
87     # IMPORT SOLUTION
88     with open(str(script.filename[0]) + str(script.filename[1]) + '_results', 'rb') as f:
89         u = pickle.Unpickler(f)
90         u.encoding = 'latin1'
91         results = u.load()
92         # results = pickle.load(f)
93     applied_force = 1 # [N]. This is also defined in Class_Script.py : def evaluate()
94
95     output = dict()
96     output['X'] = x
97     output['Cell_Size'] = script.truss.cell_size
98     output['Strut_Thickness'] = script.truss.cells[0].strut_thicknesses[0]
99     output['Truss_Name'] = script.truss.name
100    output['Pore_size'] = list()
101    for cell in script.truss.cells:
102        output['Pore_size'].append(cell.pore_size)
103
104    try: # This outputs volumetrics such as porosity, which is only possible when the solid is generated in abaqus.
105        porosity = 1 - (results['Volume'] / ((script.truss.number_of_cells * script.truss.cell_size) ** 3))
106        print("Porosity: " + str(round(porosity * 1e2, 1)) + "%")
107        print("Void Ratio: " + str(round(porosity / (1 - porosity), 1)))
108        print("Pore Sizes: ")
109        for cells in script.truss.cells:
110            for pore in cells.pore_size:
111                print(str(round(pore * 1e3, 3)) + " μm, ")
112        output['Volume'] = results['Volume'] # mm^3
113        output['Porosity'] = porosity
114        output['Void_ratio'] = porosity / (1 - porosity)
115        output['Surface_Area'] = results['Surface_Area'] # mm^2
116    except KeyError:
117        print("This Evaluation is done without calculating Porosity or Volume")
118    # Calculate Compliance Matrix
119    length_cube = (script.truss.cell_size * script.truss.number_of_cells + script.truss.cells[0].strut_thicknesses[0])
120
121    # First Quarter:
122    output['Sigma_x'] = read_stress(results, 'SIGMA_X', 'SIGMA_X_1', 0)
123    output['Sigma_y'] = read_stress(results, 'SIGMA_Y', 'SIGMA_Y_1', 1)
124    output['Sigma_z'] = read_stress(results, 'SIGMA_Z', 'SIGMA_Z_1', 2)
125    print("Elastic Modulus in X direction: " + str(round(output['Sigma_x'], 3)) + " MPa")
126    print("Elastic Modulus in Y direction: " + str(round(output['Sigma_y'], 3)) + " MPa")
127    print("Elastic Modulus in Z direction: " + str(round(output['Sigma_z'], 3)) + " MPa")
128
129    compliance_sigma = numpy.zeros([3, 3])
130    compliance_sigma[0, 0] = 1 / output['Sigma_x']
131    compliance_sigma[1, 0] = -(read_displacement(results, 'SIGMA_X', 'SIGMA_Y_1', 1) -
132                             read_displacement(results, 'SIGMA_X', 'SIGMA_Y_2_Y', 1)) * length_cube / applied_force
133    compliance_sigma[2, 0] = -(read_displacement(results, 'SIGMA_X', 'SIGMA_Z_1', 2) -
134                             read_displacement(results, 'SIGMA_X', 'SIGMA_Z_2_Z', 2)) * length_cube / applied_force
135    compliance_sigma[0, 1] = -(read_displacement(results, 'SIGMA_Y', 'SIGMA_X_1', 0) -
136                             read_displacement(results, 'SIGMA_Y', 'SIGMA_X_2_X', 0)) * length_cube / applied_force
137    compliance_sigma[1, 1] = 1 / output['Sigma_y']
138    compliance_sigma[2, 1] = -(read_displacement(results, 'SIGMA_Y', 'SIGMA_Z_1', 2) -
139                             read_displacement(results, 'SIGMA_Y', 'SIGMA_Z_2_Z', 2)) * length_cube / applied_force
140    compliance_sigma[0, 2] = -(read_displacement(results, 'SIGMA_Z', 'SIGMA_X_1', 0) -
141                             read_displacement(results, 'SIGMA_Z', 'SIGMA_X_2_X', 0)) * length_cube / applied_force
142    compliance_sigma[1, 2] = -(read_displacement(results, 'SIGMA_Z', 'SIGMA_Y_1', 1) -
143                             read_displacement(results, 'SIGMA_Z', 'SIGMA_Y_2_Y', 1)) * length_cube / applied_force
144    compliance_sigma[2, 2] = 1 / output['Sigma_z']
145
146    # Second Quarter:

```

```

147 compliance_coupling2 = numpy.zeros([3, 3])
148 compliance_coupling2[0, 0] = (read_displacement(results, 'TAU_YZ', 'SIGMA_X_1', 0) -
149     read_displacement(results, 'TAU_YZ', 'SIGMA_X_2_X', 0)) * length_cube / applied_force
150 compliance_coupling2[1, 0] = (read_displacement(results, 'TAU_YZ', 'SIGMA_Y_1', 1) -
151     read_displacement(results, 'TAU_YZ', 'SIGMA_Y_2_Y', 1)) * length_cube / applied_force
152 compliance_coupling2[2, 0] = (read_displacement(results, 'TAU_YZ', 'SIGMA_Z_1', 2) -
153     read_displacement(results, 'TAU_YZ', 'SIGMA_Z_2_Z', 2)) * length_cube / applied_force
154 compliance_coupling2[0, 1] = (read_displacement(results, 'TAU_XZ', 'SIGMA_X_1', 0) -
155     read_displacement(results, 'TAU_XZ', 'SIGMA_X_2_X', 0)) * length_cube / applied_force
156 compliance_coupling2[1, 1] = (read_displacement(results, 'TAU_XZ', 'SIGMA_Y_1', 1) -
157     read_displacement(results, 'TAU_XZ', 'SIGMA_Y_2_Y', 1)) * length_cube / applied_force
158 compliance_coupling2[2, 1] = (read_displacement(results, 'TAU_XZ', 'SIGMA_Z_1', 2) -
159     read_displacement(results, 'TAU_XZ', 'SIGMA_Z_2_Z', 2)) * length_cube / applied_force
160 compliance_coupling2[0, 2] = (read_displacement(results, 'TAU_XY', 'SIGMA_X_1', 0) -
161     read_displacement(results, 'TAU_XY', 'SIGMA_X_2_X', 0)) * length_cube / applied_force
162 compliance_coupling2[1, 2] = (read_displacement(results, 'TAU_XY', 'SIGMA_Y_1', 1) -
163     read_displacement(results, 'TAU_XY', 'SIGMA_Y_2_Y', 1)) * length_cube / applied_force
164 compliance_coupling2[2, 2] = (read_displacement(results, 'TAU_XY', 'SIGMA_Z_1', 2) -
165     read_displacement(results, 'TAU_XY', 'SIGMA_Z_2_Z', 2)) * length_cube / applied_force
166
167 # Third Quarter:
168 compliance_coupling3 = numpy.zeros([3, 3])
169 compliance_coupling3[0, 0] = (read_displacement(results, 'SIGMA_X', 'SIGMA_Z_1', 1) -
170     read_displacement(results, 'SIGMA_X', 'SIGMA_Z_2_Z', 1)) * length_cube / applied_force
171 compliance_coupling3[1, 0] = (read_displacement(results, 'SIGMA_X', 'SIGMA_X_1', 2) -
172     read_displacement(results, 'SIGMA_X', 'SIGMA_X_2_X', 2)) * length_cube / applied_force
173 compliance_coupling3[2, 0] = (read_displacement(results, 'SIGMA_X', 'SIGMA_Y_1', 0) -
174     read_displacement(results, 'SIGMA_X', 'SIGMA_Y_2_Y', 0)) * length_cube / applied_force
175 compliance_coupling3[0, 1] = (read_displacement(results, 'SIGMA_Y', 'SIGMA_Z_1', 1) -
176     read_displacement(results, 'SIGMA_Y', 'SIGMA_Z_2_Z', 1)) * length_cube / applied_force
177 compliance_coupling3[1, 1] = (read_displacement(results, 'SIGMA_Y', 'SIGMA_X_1', 2) -
178     read_displacement(results, 'SIGMA_Y', 'SIGMA_X_2_X', 2)) * length_cube / applied_force
179 compliance_coupling3[2, 1] = (read_displacement(results, 'SIGMA_Y', 'SIGMA_Y_1', 0) -
180     read_displacement(results, 'SIGMA_Y', 'SIGMA_Y_2_Y', 0)) * length_cube / applied_force
181 compliance_coupling3[0, 2] = (read_displacement(results, 'SIGMA_Z', 'SIGMA_Z_1', 1) -
182     read_displacement(results, 'SIGMA_Z', 'SIGMA_Z_2_Z', 1)) * length_cube / applied_force
183 compliance_coupling3[1, 2] = (read_displacement(results, 'SIGMA_Z', 'SIGMA_X_2_X', 2) -
184     read_displacement(results, 'SIGMA_Z', 'SIGMA_Y_2_Y', 2)) * length_cube / applied_force
185 compliance_coupling3[2, 2] = (read_displacement(results, 'SIGMA_Z', 'SIGMA_Y_1', 0) -
186     read_displacement(results, 'SIGMA_Z', 'SIGMA_Y_2_Y', 0)) * length_cube / applied_force
187
188 # Fourth Quarter:
189 output['Tau_yz'] = (read_shearing(results, 'TAU_YZ', 'SIGMA_Z_1', 1) +
190     read_shearing(results, 'TAU_YZ', 'SIGMA_Y_1', 2)) / 2
191 output['Tau_xz'] = (read_shearing(results, 'TAU_XZ', 'SIGMA_X_1', 2) +
192     read_shearing(results, 'TAU_XZ', 'SIGMA_Z_1', 0)) / 2
193 output['Tau_xy'] = (read_shearing(results, 'TAU_XY', 'SIGMA_Y_1', 0) +
194     read_shearing(results, 'TAU_XY', 'SIGMA_X_1', 1)) / 2
195 print("Shearing Modulus in yz direction: " + str(round(output['Tau_yz'], 3)) + " MPa")
196 print("Shearing Modulus in xz direction: " + str(round(output['Tau_xz'], 3)) + " MPa")
197 print("Shearing Modulus in xy direction: " + str(round(output['Tau_xy'], 3)) + " MPa")
198
199 compliance_tau = numpy.zeros([3, 3])
200 compliance_tau[0, 0] = 1 / output['Tau_yz']
201 compliance_tau[1, 0] = (read_displacement(results, 'TAU_YZ', 'SIGMA_Z_1', 0) -
202     read_displacement(results, 'TAU_YZ', 'SIGMA_Z_2_Z', 0)) * length_cube / applied_force
203 compliance_tau[2, 0] = (read_displacement(results, 'TAU_YZ', 'SIGMA_Y_1', 0) -
204     read_displacement(results, 'TAU_YZ', 'SIGMA_Y_2_Y', 0)) * length_cube / applied_force
205 compliance_tau[0, 1] = (read_displacement(results, 'TAU_XZ', 'SIGMA_Z_1', 1) -
206     read_displacement(results, 'TAU_XZ', 'SIGMA_Z_2_Z', 1)) * length_cube / applied_force
207 compliance_tau[1, 1] = 1 / output['Tau_xz']
208 compliance_tau[2, 1] = (read_displacement(results, 'TAU_XZ', 'SIGMA_X_1', 1) -
209     read_displacement(results, 'TAU_XZ', 'SIGMA_X_2_X', 1)) * length_cube / applied_force
210 compliance_tau[0, 2] = (read_displacement(results, 'TAU_XY', 'SIGMA_Y_1', 2) -
211     read_displacement(results, 'TAU_XY', 'SIGMA_Y_2_Y', 2)) * length_cube / applied_force
212 compliance_tau[1, 2] = (read_displacement(results, 'TAU_XY', 'SIGMA_X_1', 2) -
213     read_displacement(results, 'TAU_XY', 'SIGMA_X_2_X', 2)) * length_cube / applied_force
214 compliance_tau[2, 2] = 1 / output['Tau_xy']
215
216 output['Compliance'] = numpy.zeros([6, 6])
217 output['Compliance'][0:3, 0:3] = compliance_sigma
218 output['Compliance'][0:3, 3:7] = compliance_coupling2
219 output['Compliance'][3:7, 0:3] = compliance_coupling3
220 output['Compliance'][3:7, 3:7] = compliance_tau
221

```

```

222     # POISSON'S RATIO
223     output['v21'] = -output['Compliance'][1, 0] / output['Compliance'][0, 0]
224     output['v31'] = -output['Compliance'][2, 0] / output['Compliance'][0, 0]
225     output['v32'] = -output['Compliance'][1, 2] / output['Compliance'][2, 2]
226     print("Poisson's ratio v21: " + str(round(output['v21'], 3)))
227     print("Poisson's ratio v31: " + str(round(output['v31'], 3)))
228     print("Poisson's ratio v32: " + str(round(output['v32'], 3)))
229
230     print("Compliance Matrix [1/MPa]: ")
231     numpy.set_printoptions(precision=2, suppress=True)
232     print(numpy.multiply(output['Compliance'], 1e3))
233
234
235
236     # APPENDS RESULTS TO A CSV FILE AND A SERIALIZED PICKLE FILE
237     def append_output_to_file(options, output, output_file):
238         result_file = open(output_file, 'a')
239         if options['output']['Step']:
240             result_file.write(str(output['Step']) + ", ")
241         if options['output']['Truss_Name']:
242             result_file.write(str(output['Truss_Name']) + ", ")
243         if options['output']['Fitness']:
244             result_file.write(str(round(output['Fitness'], 3)) + ", ")
245         if options['output']['Cell_size']:
246             result_file.write(str(round(output['Cell_Size'], 3)) + ", ")
247         if options['output']['Strut_Thickness']:
248             result_file.write(str(round(output['Strut_Thickness'] * 1e3, 3)) + ", ")
249         if options['output']['Pore_size']:
250             counter = 0
251             for cells in output['Pore_size']:
252                 for pore in cells:
253                     result_file.write(str(round(pore * 1e3, 3)) + ", ")
254                     counter += 1
255                 for i in range(0, 4 - counter):
256                     result_file.write("None, ")
257         if options['output']['Young's Modulus']:
258             result_file.write(str(round(output['Sigma_z'], 3)) + ", " +
259                             str(round(output['Sigma_y'], 3)) + ", " +
260                             str(round(output['Sigma_x'], 3)) + ", ")
261         if options['output']['Shearing Modulus']:
262             result_file.write(str(round(output['Tau_yz'], 3)) + ", " +
263                             str(round(output['Tau_xz'], 3)) + ", " +
264                             str(round(output['Tau_xy'], 3)) + ", ")
265         if options['output']['Poisson's Ratio']:
266             result_file.write(str(round(output['v21'], 3)) + ", " +
267                             str(round(output['v31'], 3)) + ", " +
268                             str(round(output['v32'], 3)) + ", ")
269         try:
270             if options['output']['Volume']:
271                 result_file.write(str(round(output['Volume'], 3)) + ", ")
272             if options['output']['Porosity']:
273                 result_file.write(str(round(output['Porosity'] * 100, 3)) + ", ")
274             if options['output']['Void_ratio']:
275                 result_file.write(str(round(output['Void_ratio'] * 1, 3)) + ", ")
276             if options['output']['Surface_Area']:
277                 result_file.write(str(round(output['Surface_Area'], 3)) + ", ")
278         except KeyError:
279             if options['output']['Volume']:
280                 result_file.write("None, ")
281             if options['output']['Porosity']:
282                 result_file.write("None, ")
283             if options['output']['Void_ratio']:
284                 result_file.write("None, ")
285             if options['output']['Surface_Area']:
286                 result_file.write("None, ")
287         if options['output']['X']:
288             result_file.write(str(output['X']))
289         result_file.write('\n')
290         result_file.close()
291
292     # # APPENDS OUTPUT TO THE PICKLED OUTPUT
293     unpickle = pickle._Unpickler(open(str(output_file[:len(output_file) - 4]) + "_pickle", 'rb'))
294     unpickle.encoding = 'latin1'
295     output_old = unpickle.load()
296     output_old.append(output)

```

```

297
298     file = open(str(output_file[:len(output_file) - 4]) + "_pickle", 'wb')
299     pickle.dump(output_old, file)
300     file.close()
301
302
303     # SAFES THE INPUT IN A SERIALIZED PICKLE FILE
304     def pickle_input(x, truss_name, input_file):
305         pickle_dump = list()
306         pickle_dump.append(x)
307         pickle_dump.append(truss_name)
308         file = open(str(input_file), 'wb')
309         pickle.dump(pickle_dump, file)
310         file.close()
311
312
313     universal_counter = 0
314
315
316 ##### BEGIN OF function #####
317 # BEGIN OF function
318
319
320     def objective_function(inputs, options):
321         global universal_counter
322         universal_counter += 1 # Careful this is global
323
324         # PRINT INFO FROM INPUT
325         print("Counter: " + str(universal_counter))
326         print("Truss: " + str(inputs['truss_name']))
327         print("Strut Minimal Thickness: " + str(inputs['strut_min_thickness']))
328         print("Strut Thickness Multiplier: " + str(inputs['strut_thickness_multiplicator']))
329         print("Number of Cells: " + str(inputs['number_of_cells']))
330         print("Cell Size: " + str(round(inputs['cell_size'], 3)) + " mm")
331         print("Total Size: " + str(round(inputs['number_of_cells'] * inputs['cell_size'], 3)) + " mm")
332         print("Cell Ratio: " + str(inputs['cell_ratio']) + "\n\n\n")
333
334         # MULTIPLY strut_thickesses WITH min_thickness
335         thicknesses = list()
336         fitness = 0
337         for multiplicator in inputs['strut_thickness_multiplicator']:
338             if multiplicator > 0 and inputs['strut_min_thickness'] > 0:
339                 thicknesses.append(inputs['strut_min_thickness'] * multiplicator)
340             else: # Blocks negative values for strut thicknesses
341                 thicknesses.append(abs(inputs['strut_min_thickness'] * multiplicator))
342             fitness += 1e9 * abs(multiplicator) * abs(inputs['strut_min_thickness'])
343
344         filename = list()
345         filename.append(inputs['calculating_directory'])
346         filename.append(inputs['job_name'] + str(universal_counter))
347         filename.append(".py")
348         # GENERATE TRUSS
349         truss = generate_truss(truss_name=inputs['truss_name'], affix="", cell_size=inputs['cell_size'],
350                               strut_thickesses=thicknesses, number_of_cells=inputs['number_of_cells'],
351                               cell_ratio=inputs['cell_ratio'])
352
353         # GENERATE SCRIPT
354
355         # initialize.
356         model_script = Script(filename=filename, truss=truss, material=inputs['material'],
357                               abaqus_path=options['abaqus_path'], abaqus_version=options['abaqus_version'])
358         # generate wireframe and evaluate.
359         model_script.evaluate(create_steps=options['create_steps'], submit_job=options['submit_job'],
360                               read_output=options['read_output'], number_of_cells=inputs['number_of_cells'])
361         # generate solid and export to stl.
362         if options['stl_generate']:
363             model_script.generate_solid(strut_name=options['strut_cross_section'], cutoff=options['cutoff'],
364                                         number_of_cells=inputs['number_of_cells'])
365             model_script.export_stl()
366
367         # saves the results from the simulation in a serialized pickle file.
368         model_script.pickle_dump()
369
370         # RUN SCRIPT
371         run(script=model_script, gui=options['gui'])
372

```

```

372 # OPEN VIEWERS TO SEE RESULT
373 if options['stl_view']:
374     stl_viewer(model_script)
375 if options['odb_view']:
376     odb_viewer(model_script)
377
378 # CALCULATE COMPLIANCE FROM DISPLACEMENTS FROM THE SIMULATION
379 if options['read_output']:
380     output = read_results(model_script, inputs['strut_thickness_multiplicator'])
381 else:
382     output = dict()
383
384 # CALCULATE FITNESS
385 if options['method'] == 'optimization':
386     for variable in options['fitness_variables']:
387         fitness += (abs(output[variable] - options['fitness_variables'][variable][0]) *
388                      options['fitness_variables'][variable][1]) * options['fitness_variables'][variable][2]
389     fitness /= len(options['fitness_variables'])
390
391 # SAFE OUTPUT INTO CSV FILE
392 if options['read_output']:
393     output['Step'] = universal_counter
394     output['Fitness'] = fitness
395     append_output_to_file(options, output, inputs['output_file'] + '.csv')
396
397 # SAFE INPUT VALUES AS A SERIALIZED PICKLE FILE
398 pickle_input(inputs['strut_thickness_multiplicator'],
399               inputs['truss_name'],
400               str(inputs['calculating_directory']) +
401               str(inputs['job_name']) +
402               str(universal_counter) +
403               "_input")
404
405 # PLOT FITNESS
406 if options['plot_fitness']:
407     pyplot.figure(1)
408     pyplot.subplot(311)
409     pyplot.xlabel('Step')
410     pyplot.ylabel('Fitness')
411     pyplot.ylim((100, 2000))
412     pyplot.scatter(universal_counter, fitness)
413     pyplot.legend()
414     pyplot.subplot(312)
415     pyplot.xlabel('Step')
416     pyplot.ylabel('Fitness')
417     pyplot.ylim((20, 100))
418     pyplot.scatter(universal_counter, fitness)
419     pyplot.legend()
420     pyplot.subplot(313)
421     pyplot.xlabel('Step')
422     pyplot.ylabel('Fitness')
423     pyplot.ylim((-5, 20))
424     pyplot.scatter(universal_counter, fitness)
425     pyplot.legend()
426     pyplot.show()
427     pyplot.pause(0.00001)
428
429 print("Fitness: " + str(fitness))
430 print("#####")
431 return fitness

```

A.4.3 *Class_Script.py*

```

1    """
2    This File initiates the abaqus script to run for the optimization.
3    It consists of several functions that append text to a .py -file which is then run in abaqus.
4
5    The first block of functions creates a solid model which can be exported to stl.
6    The second block of functions creates a wireframe model and does an analysis of its properties.
7    The third block reads the output database that the simulation creates and returns it in a pickled file.
8
9    """
10   import sys
11   import math
12   import numpy
13
14
15   ######
16   # FUNCTIONS FOR THE SOLID AND STL-FILE-GENERATION
17   # EXTRUDES A SIMPLE STRUT. POSSIBLE SECTIONS ARE SQUARE, HEXAGON, OCTAGON, DODECAGON
18   def generate_strut(filename, strut_name, affix, connection):
19       file = open("".join(filename), "a")
20       thickness = connection[2]
21       begin_vector = numpy.asarray(connection[0])
22       end_vector = numpy.asarray(connection[1])
23       distance = numpy.subtract(end_vector, begin_vector)
24       length = numpy.linalg.norm(distance)
25
26       # GENERATE BASE SKETCH
27       file.write(
28           "s = mdb.models['Model-1'].ConstrainedSketch(name='__profile__', sheetSize=200.0)\n"
29           "g, v, d, c = s.geometry, s.vertices, s.dimensions, s.constraints\n"
30           "s.setPrimaryObject(option=STANDALONE)\n"
31
32       # SQUARE
33       if strut_name == "square":
34           file.write(
35               "s.Line(point1=(" + str(-thickness / 2) + ", " + str(-thickness / 2) + "), point2=(" + str(thickness / 2) +
36               ", " + str(-thickness / 2) + "))\n"
37               "s.Line(point1=(" + str(thickness / 2) + ", " + str(-thickness / 2) + "), point2=(" + str(thickness / 2) +
38               ", " + str(thickness / 2) + "))\n"
39               "s.Line(point1=(" + str(thickness / 2) + ", " + str(thickness / 2) + "), point2=(" + str(-thickness / 2) +
40               ", " + str(thickness / 2) + "))\n"
41               "s.Line(point1=(" + str(-thickness / 2) + ", " + str(thickness / 2) + "), point2=(" + str(-thickness / 2) +
42               ", " + str(-thickness / 2) + "))\n"
43
44       # OCTAGON
45       elif strut_name == "octagon":
46           b = math.sqrt(2) / 2 * (1 - (math.sqrt(2) / (1 + math.sqrt(2)))) * thickness
47           file.write(
48               "s.Line(point1=(" + str(-thickness / 2 + b) + ", " + str(-thickness / 2) + "), point2=(" + str(thickness /
49               - thickness / 2 - b) + ", " + str(-thickness / 2) + "))\n"
50               "s.Line(point1=(" + str(thickness / 2 - b) + ", " + str(-thickness / 2) + "), point2=(" + str(thickness /
51               - thickness / 2 + b) + ", " + str(-thickness / 2) + "))\n"
52               "s.Line(point1=(" + str(thickness / 2) + ", " + str(-thickness / 2 + b) + "), point2=(" + str(thickness /
53               - thickness / 2) + ", " + str(thickness / 2 - b) + "))\n"
54               "s.Line(point1=(" + str(thickness / 2) + ", " + str(thickness / 2 - b) + "), point2=(" + str(thickness /
55               - thickness / 2 - b) + ", " + str(thickness / 2) + "))\n"
56               "s.Line(point1=(" + str(thickness / 2 - b) + ", " + str(thickness / 2) + "), point2=(" + str(-thickness /
57               / 2 + b) + ", " + str(thickness / 2) + "))\n"
58               "s.Line(point1=(" + str(-thickness / 2 + b) + ", " + str(thickness / 2) + "), point2=(" + str(-thickness /
59               / 2) + ", " + str(thickness / 2 - b) + "))\n"
60               "s.Line(point1=(" + str(-thickness / 2) + ", " + str(thickness / 2 - b) + "), point2=(" + str(-thickness /
61               / 2) + ", " + str(-thickness / 2 + b) + "))\n"
62               "s.Line(point1=(" + str(-thickness / 2) + ", " + str(-thickness / 2 + b) + "), point2=(" + str(-thickness /
63               / 2 + b) + ", " + str(-thickness / 2) + "))\n"
64
65       # DODECAGON
66       elif strut_name == "dodecagon":
67           for counter in range(0, 12):
68               file.write("s.Line(point1=(" +
69                   str(math.cos((15 + 30 * counter) / 180 * math.pi) * thickness / 2) + ", " +
70                   str(math.sin((15 + 30 * counter) / 180 * math.pi) * thickness / 2) + "), point2=(" +
71                   str(math.cos((15 + 30 * (counter + 1)) / 180 * math.pi) * thickness / 2) + ", " +

```

```

72             str(math.sin((15 + 30 * (counter + 1)) / 180 * math.pi) * thickness / 2) + ")\n")
73
74 # HEXAGON
75 elif strut_name == "hexagon":
76     for counter in range(0, 6):
77         file.write("s.Line(point1=" +
78                     str(math.cos((30 + 60 * counter) / 180 * math.pi) * thickness / 2) + ", " +
79                     str(math.sin((30 + 60 * counter) / 180 * math.pi) * thickness / 2) + ", point2=" +
80                     str(math.cos((30 + 60 * (counter + 1)) / 180 * math.pi) * thickness / 2) + ", " +
81                     str(math.sin((30 + 60 * (counter + 1)) / 180 * math.pi) * thickness / 2) + ")\n")
82
83 else:
84     sys.exit("ERROR: strut_name specified is not defined in the library")
85
86 file.write("mdb.models['Model-1'].ConstrainedSketch(name='strut_sketch-' + str(strut_name) +
87             str(affix) + ', objectToCopy=s)\n")
88
89 # EXTRUDE BASE SKETCH
90 file.write(
91     "p = mdb.models['Model-1'].Part(name='strut_" + str(strut_name) + str(affix) + +
92     "', dimensionality=THREE_D, type=DEFORMABLE_BODY)\n"
93     "p = mdb.models['Model-1'].parts['strut_" + str(strut_name) + str(affix) + "']\n"
94     "p.BaseSolidExtrude(sketch=s, depth=" + str(length) + ")\n"
95     "s.unsetPrimaryObject()\n")
96
97 file.close()
98
99
100 # ADDS A SINGLE STRUT TO THE CELL STRUCTURE. THIS IS USED IN generate_cell()
101 def add_strut_to_cell(filename, strut_name, instance_name, strut_counter, connections):
102     file = open("".join(filename), "a")
103     begin_vector = numpy.asarray(connections[0]) # Determine the rotation and translation of the strut
104     end_vector = numpy.asarray(connections[1])
105     distance = numpy.subtract(end_vector, begin_vector)
106     length = numpy.linalg.norm(distance)
107     normal = numpy.cross(distance, [0, 0, 1])
108     if numpy.linalg.norm(normal) == 0:
109         normal = [1, 0, 0]
110     angle = -abs(math.acos(numpy.dot([0, 0, 1], distance) / length)) / math.pi * 180
111     # INCLUDE INTO ASSEMBLY
112     file.write(
113         "a = mdb.models['Model-1'].rootAssembly\n"
114
115         "a.DatumCsysByDefault(CARTESIAN)\n"
116
117         "p = mdb.models['Model-1'].parts['strut_" + str(strut_name) + str(strut_counter) + "']\n"
118
119         "a.Instance(name='instance_" + str(instance_name) + "-" + str(strut_counter) + "", part=p, dependent=ON)\n"
120
121         "a = mdb.models['Model-1'].rootAssembly\n"
122
123         "a.rotate(instanceList='instance_" + str(instance_name) + "-" + str(strut_counter) + +
124             "', axisPoint=(0.0, 0.0, 0.0), axisDirection=" + str(normal[0]) + ", " +
125             str(normal[1]) + ", " + str(normal[2]) + ", angle=" + str(angle) + ")\n"
126
127         "a.translate(instanceList='instance_" + str(instance_name) + "-" + str(strut_counter) + "", ), vector=" +
128             str(begin_vector[0]) + ", " +
129             str(begin_vector[1]) + ", " +
130             str(begin_vector[2]) + ")\n"
131     )
132
133     file.close()
134
135
136 # ADDS A SINGLE CELL TO THE TRUSS STRUCTURE. THIS IS USED IN generate_solid()
137 def add_cell_to_assembly(filename, cell_name, instance_name, cell_counter, truss_node_coordinates):
138     file = open("".join(filename), "a")
139
140     # INCLUDE INTO ASSEMBLY
141     file.write(
142         "a = mdb.models['Model-1'].rootAssembly\n"
143
144         "a.DatumCsysByDefault(CARTESIAN)\n"
145
146         "p = mdb.models['Model-1'].parts['" + str(cell_name) + "']\n"

```

```

147
148     "a.Instance(name='instance_" + str(instance_name) + "-" + str(cell_counter) + "'", part=p, dependent=ON)\n"
149
150     "a = mdb.models['Model-1'].rootAssembly\n"
151
152     "a.translate(instanceList=('instance_" + str(instance_name) + "-" + str(cell_counter) + "'", ), vector=("
153     str(truss_node_coordinates[0]) + ", " +
154     str(truss_node_coordinates[1]) + ", " +
155     str(truss_node_coordinates[2]) + "))\n"
156 file.close()
157
158
159 # MERGES ALL PARTS OF AN INSTANCE. THIS IS USED IN generate_cell() AND generate_solid()
160 def merge(filename, merge_name, instance_name, part_counter):
161     file = open("".join(filename), "a")
162     if part_counter > 1:
163         file.write("a.InstanceFromBooleanMerge(name='" + str(merge_name) + "'", instances="")
164         for counter in range(0, part_counter):
165             file.write("a.instances['instance_" + str(instance_name) + "-" + str(counter) + "'], ")
166         file.write(")", originalInstances=SUPPRESS, domain=GEOOMETRY)\n")
167     for counter in range(0, part_counter):
168         file.write("del a.features['instance_" + str(instance_name) + "-" + str(counter) + "']\n")
169     else:
170         file.write("mdb.models['Model-1'].rootAssembly.features.changeKey(fromName='instance_" + str(instance_name) +
171                     "-0', toName='" + str(merge_name) + "-1')\n")
172 file.close()
173
174
175 # GENERATES THE SOLID CELL
176 def generate_cell(filename, cell, strut_name):
177     strut_counter = 0
178     for connection in cell.connections:
179         generate_strut(filename, strut_name, strut_counter, connection)
180         add_strut_to_cell(filename, strut_name, cell.name, strut_counter, connection)
181     strut_counter += 1
182
183 merge(filename, cell.name, cell.name, strut_counter) # "Instance with name 'cell.name'"
184 delete_struts(filename, strut_name, strut_counter)
185
186
187 # DELETES ALL CELL INSTANCES AFTER MERGING
188 def delete_struts(filename, strut_name, number_of_struts):
189     for strut_counter in range(0, number_of_struts):
190         file = open("".join(filename), "a")
191         file.write("del mdb.models['Model-1'].parts['strut_" + str(strut_name) + str(strut_counter) + "']\n")
192         file.close()
193
194
195 def delete_instances(filename, objects):
196     file = open("".join(filename), "a")
197     for obj in objects:
198         file.write("del a.features['" + str(obj.name) + "-1']\n")
199
200 file.close()
201
202
203 # CUT OFF THE BORDERS OF THE TRUSS
204 def cut_off_borders(filename, cutoff_name, merge_name, truss, number_of_cells):
205     file = open("".join(filename), "a")
206     file.write("s = mdb.models['Model-1'].ConstrainedSketch(name='__profile__', sheetSize=200.0)\n"
207     "g, v, d, c = s.geometry, s.vertices, s.dimensions, s.constraints\n"
208     "s.setPrimaryObject(option=STANDALONE)\n"
209     file.write("s.Line(point1=(" + str(-(number_of_cells + 4) * truss.cell_size / 2) + ", " +
210                 str(-(number_of_cells + 4) * truss.cell_size / 2) + "), point2=(" +
211                 str((number_of_cells + 4) * truss.cell_size / 2) + ", " +
212                 str(-(number_of_cells + 4) * truss.cell_size / 2) + "))\n"
213     "s.Line(point1=(" + str((number_of_cells + 4) * truss.cell_size / 2) + ", " +
214                 str(-(number_of_cells + 4) * truss.cell_size / 2) + "), point2=(" +
215                 str((number_of_cells + 4) * truss.cell_size / 2) + ", " +
216                 str((number_of_cells + 4) * truss.cell_size / 2) + "))\n"
217     "s.Line(point1=(" + str((number_of_cells + 4) * truss.cell_size / 2) + ", " +
218                 str((number_of_cells + 4) * truss.cell_size / 2) + "), point2=(" +
219                 str(-(number_of_cells + 4) * truss.cell_size / 2) + ", " +
220                 str((number_of_cells + 4) * truss.cell_size / 2) + "))\n"
221     "s.Line(point1=(" + str(-(number_of_cells + 4) * truss.cell_size / 2) + ", " +

```

```

222         str((number_of_cells + 4) * truss.cell_size / 2) + "), point2=" +
223         str(-(number_of_cells + 4) * truss.cell_size / 2) + ", " +
224         str(-(number_of_cells + 4) * truss.cell_size / 2) + "))\n")
225     file.write("mdb.models['Model-1'].ConstrainedSketch(name='cut_off', objectToCopy=s)\n")
226     file.write(
227         "p = mdb.models['Model-1'].Part(name='cut_off', dimensionality=THREE_D, type=DEFORMABLE_BODY)\n"
228         "p = mdb.models['Model-1'].parts['cut_off']\n"
229         "p.BaseSolidExtrude(sketch=s, depth=" + str(2 * truss.cell_size) + ")\\n"
230         "s.unsetPrimaryObject()\n")
231     file.write(
232         "a = mdb.models['Model-1'].rootAssembly\n"
233
234         "a.DatumCsysByDefault(CARTESIAN)\n"
235
236         "p = mdb.models['Model-1'].parts['cut_off']\n"
237
238         "a.Instance(name='instance_cut_off1', part=p, dependent=ON)\n"
239         "a.Instance(name='instance_cut_off2', part=p, dependent=ON)\n"
240         "a.Instance(name='instance_cut_off3', part=p, dependent=ON)\n"
241         "a.Instance(name='instance_cut_off4', part=p, dependent=ON)\n"
242         "a.Instance(name='instance_cut_off5', part=p, dependent=ON)\n"
243         "a.Instance(name='instance_cut_off6', part=p, dependent=ON)\n"
244
245         "a.translate(instanceList=('instance_cut_off1', ), vector=(" +
246             str(0) + ", " +
247             str(0) + ", " +
248             str((number_of_cells - 0.5) * truss.cell_size) + "))\n"
249         "a.translate(instanceList=('instance_cut_off1', ), vector=(" + str(truss.cell_size) + "," +
250             str(truss.cell_size) + ",0))\n"
251         "a.rotate(instanceList=('instance_cut_off2', ), axisPoint=(0.0, 0.0, 0.0), axisDirection=(0,1,0), angle=180)\n"
252         "a.translate(instanceList=('instance_cut_off2', ), vector=(" +
253             str(0) + ", " +
254             str(0) + ", " +
255             str(-0.5 * truss.cell_size) + "))\n"
256             "a.translate(instanceList=('instance_cut_off2', ), vector=(" +
257                 str(truss.cell_size) + "," + str(truss.cell_size) + ",0))\n"
258             "a.rotate(instanceList=('instance_cut_off3', ), axisPoint=(0.0, 0.0, 0.0), axisDirection=(0,1,0), angle=-90)\n"
259             "a.translate(instanceList=('instance_cut_off3', ), vector=(" +
260                 str(-0.5 * truss.cell_size) + ", " +
261                 str(0) + ", " +
262                 str(0) + "))\n"
263                 "a.translate(instanceList=('instance_cut_off3', ), vector=(0, " + str(truss.cell_size) + "," +
264                     str(truss.cell_size) + "))\n"
265                 "a.rotate(instanceList=('instance_cut_off4', ), axisPoint=(0.0, 0.0, 0.0), axisDirection=(0,1,0), angle=90)\n"
266                 "a.translate(instanceList=('instance_cut_off4', ), vector=(" +
267                     str((number_of_cells - 0.5) * truss.cell_size) + ", " +
268                     str(0) + ", " +
269                     str(0) + "))\n"
270                     "a.translate(instanceList=('instance_cut_off4', ), vector=(0, " + str(truss.cell_size) + "," +
271                         str(truss.cell_size) + "))\n"
272                     "a.rotate(instanceList=('instance_cut_off5', ), axisPoint=(0.0, 0.0, 0.0), axisDirection=(1,0,0), angle=90)\n"
273                     "a.translate(instanceList=('instance_cut_off5', ), vector=(" +
274                         str(0) + ", " +
275                         str(-0.5 * truss.cell_size) + ", " +
276                         str(0) + "))\n"
277                     "a.translate(instanceList=('instance_cut_off5', ), vector=(" + str(truss.cell_size) + ", 0, " +
278                         str(truss.cell_size) + "))\n"
279                     "a.rotate(instanceList=('instance_cut_off6', ), axisPoint=(0.0, 0.0, 0.0), axisDirection=(1,0,0), angle=-90)\n"
280                     "a.translate(instanceList=('instance_cut_off6', ), vector=(" +
281                         str(0) + ", " +
282                         str((number_of_cells - 0.5) * truss.cell_size) + ", " +
283                         str(0) + "))\n"
284                     "a.translate(instanceList=('instance_cut_off6', ), vector=(" + str(truss.cell_size) + ", 0, " +
285                         str(truss.cell_size) + "))\n"
286     )
287
288     file.write("a.InstanceFromBooleanCut(name='" + cutoff_name + "', "
289         "instanceToBeCut=mdb.models['Model-1'].rootAssembly.instances['" + merge_name + "'], "
290         "cuttingInstances=(a.instances['instance_cut_off1'], a.instances['instance_cut_off2'], "
291         "a.instances['instance_cut_off3'], a.instances['instance_cut_off4'], a.instances['instance_cut_off5'], "
292         "a.instances['instance_cut_off6'], ), originalInstances=DELETE)\n")
293     file.close()
294
295
296 # MEASURE THE VOLUME AND SURFACE ARE OF THE TRUSS CREATED IN ABAQUS. ONLY WORKS IF THE BORDERS ARE CUT OFF

```

```

297 def get_area(script, filename, cutoff):
298     file = open("".join(filename), "a")
299     file.write("results['Volume'] = a.getVolume()\n"
300             "results['Surface_Area'] = a.getArea(a.instances['" + filename[1] +
301             "-1'].faces)\n"
302             "print('Results: ' + str(results['Surface_Area']))\n")
303     if cutoff:
304         if script.truss.name == "cubes" or script.truss.name == "body_centered_cubes" or \
305             script.truss.name == "octetrahedrons" or script.truss.name == "face_diagonal_cubes_alt" or \
306             script.truss.name == "void_octetrahedrons":
307             file.write("area_error=-6 * a.getArea(a.instances['" + filename[1] +
308             "-1'].faces.findAt((" + str(script.truss.cells[0].strut_thicknesses[0] / 32 -
309             script.truss.cell_size / 2) +
310             ", " + str(-script.truss.cell_size / 2) + ", " +
311             str(script.truss.cells[0].strut_thicknesses[0] / 32 - script.truss.cell_size / 2) + ") ,))\n"
312             "print('Area Error: ' + str(area_error))\n"
313             "results['Surface_Area']=area_error\n")
314     elif script.truss.name == "diamonds":
315         file.write("-12 * a.getArea(a.instances['" + filename[1] +
316             "-1'].faces.findAt((" +
317             str(script.truss.cells[0].strut_thicknesses[0] / 32 - script.truss.cell_size / 2) +
318             ", " + str(-script.truss.cell_size / 2) + ", " +
319             str(script.truss.cells[0].strut_thicknesses[0] / 32 - script.truss.cell_size / 2) + ") ,))\n"
320         )
321         file.write("-6 * a.getArea(a.instances['" + filename[1] +
322             "-1'].faces.findAt((" + str(0) +
323             ", " + str(-script.truss.cell_size / 2) + ", " + str(0) + ") ,))\n"
324         )
325     elif script.truss.name == "truncated_cubes":
326         file.write("-6 * a.getArea(a.instances['" + filename[1] +
327             "-1'].faces.findAt((" +
328             str(script.truss.cells[0].strut_thicknesses[0] / 32 - script.truss.cell_size / 2) +
329             ", " + str(-script.truss.cell_size / 2) + ", " + str(0) + ") ,))\n"
330         )
331     else:
332         print("The Calculated Surface Area is too high for it does not subtract the outside area of the " +
333             str(script.truss.name) + " truss")
334     else:
335         print("No surface area correction because there is no cutoff")
336
337 ##### FUNCTIONS FOR THE FINITE ELEMENT ANALYSIS AND WIREFRAME GENERATION
338
339
340
341 # GENERATE THE WIREFRAME
342 def generate_wireframe(filename, truss):
343     file = open("".join(filename), "a")
344     file.write(
345         "p = mdb.models['Model-1'].Part(name='Part-1', dimensionality=THREE_D, type=DEFORMABLE_BODY)\n"
346         "p.ReferencePoint(point=(0.0, 0.0, 0.0))\n"
347         "p.WirePolyLine(points=(\n")
348     for nodes in truss.nodes:
349         for connections in nodes[3].connections:
350             file.write("                (" + str(connections[0][0] + nodes[0]) +
351                         ", " + str(connections[0][1] + nodes[1]) +
352                         ", " + str(connections[0][2] + nodes[2]) +
353                         "), (" + str(connections[1][0] + nodes[0]) +
354                         ", " + str(connections[1][1] + nodes[1]) +
355                         ", " + str(connections[1][2] + nodes[2]) +
356                         ")), \n")
357
358     file.write(
359         "), mergeType=IMPRINT, meshable=ON)\n"
360         "p = mdb.models['Model-1'].parts['Part-1']\n"
361         "e = p.edges\n"
362         "edges = e.getSequenceFromMask(mask='[#7 ]', ) , )\n"
363         "p.Set(edges=edges, name='Wire-' + str(truss.name) + '1')\n"
364         "a = mdb.models['Model-1'].rootAssembly\n"
365         "a.DatumCsysByDefault(CARTESIAN)\n"
366     file.close()
367
368
369 # DEFINE THE MATERIAL PROPERTIES
370 def define_material(filename, material_name, young_modulus, poisson_ratio):
371     file = open("".join(filename), "a")

```

```

372     file.write("mdb.models['Model-1'].Material(name='" + str(material_name) +
373                 "', description='Automatically generated. See Class_Script')\n"
374                 "mdb.models['Model-1'].materials['" + str(material_name) + "'].Elastic(table=(( " +
375                     str(young_modulus) + ", " + str(poission_ratio) + "), ))\n")
376     file.close()
377
378
379 # ASSIGNS PROFILE AND MATERIAL
380 def assign_material_to_wire(filename, truss, material_name):
381     file = open("", join(filename), "a")
382
383     for cell in truss.cells:
384
385         list_of_nodes = list()
386         for node in truss.nodes:
387             if cell.name == node[3].name:
388                 list_of_nodes.append(node)
389
390         counter = 0
391         for connection in cell.connections:
392             file.write("mdb.models['Model-1'].CircularProfile(name='Profile-" + str(cell.name) + "-" + str(counter) + "
393                         "", r=" + str(connection[2] / 2) + ")\n"
394                         "mdb.models['Model-1'].BeamSection(name='Section-" + str(cell.name) + "-" + str(counter) + "
395                         "", integration=DURING_ANALYSIS, poissonRatio=0.0, profile='Profile-" + str(cell.name) + "-" +
396                         str(counter) + "', material='" + str(material_name) + "', temperatureVar=LINEAR,
397                         "consistentMassMatrix=False)\n"
398                         "r = regionToolset.Region(edges=p.edges.findAt("
399                         for connection_node in list_of_nodes:
400                             file.write("((" + str(1 / 2 * (connection[0][0] + connection[1][0]) + connection_node[0]) + ", "
401                                         str(1 / 2 * (connection[0][1] + connection[1][1]) + connection_node[1]) + ", "
402                                         str(1 / 2 * (connection[0][2] + connection[1][2]) + connection_node[2]) + ", "
403                                         ")), ")
404                         file.write("))\n"
405                         "p.SectionAssignment(region=r, sectionName='Section-" + str(cell.name) + "-" + str(counter) + "
406                         "", offset=0.0,"
407                         "offsetType=MIDDLE_SURFACE, offsetField='",
408                         "thicknessAssignment=FROM_SECTION)\n"
409                         "p.assignBeamSectionOrientation(region=r, method=N1_COSINES, ni=(5,7,3))\n"
410
411         counter += 1
412     del counter
413
414 # MESHES THE TRUSS
415 def mesh_part(filename, mesh_size):
416     file = open("", join(filename), "a")
417     file.write("a = mdb.models['Model-1'].rootAssembly\n"
418                 "p = mdb.models['Model-1'].parts['Part-1']\n"
419                 "a.Instance(name='Part-1-1', part=p, dependent=ON)\n"
420                 "p.seedPart(size=" + str(mesh_size) + ", deviationFactor=0.1, minSizeFactor=0.1)\n"
421                 "p.generateMesh()\n"
422                 "p.setElementType(regions=regionToolset.Region(p.edges), "
423                 "elemTypes=(mesh.ElemType(elemCode=B32, elemLibrary=STANDARD),))\n" # B32
424     file.close()
425
426
427 # CREATES A STATIC STEP
428 def create_static_step(filename, step_name):
429     file = open("", join(filename), "a")
430     file.write("mdb.models['Model-1'].StaticStep(name='" + str(step_name) + "', previous='Initial')\n")
431     file.close()
432
433
434 # DEFINES LOADINGS ON THE TRUSS
435 def define_loadings(filename, step_name, affix, vertices, load_type, load_type_letter, force):
436     file = open("", join(filename), "a")
437     file.write("loading_points = a.instances['Part-1-1'].vertices.findAt()")
438     for vertex_coordinates in vertices:
439         file.write("((" + str(vertex_coordinates[0]) +
440                     ", " + str(vertex_coordinates[1]) +
441                     ", " + str(vertex_coordinates[2]) + "), ), ")
442     file.write(
443         "\n"
444         "mdb.models['Model-1']."' + str(load_type) + "(name='" + str(step_name + affix) + "_load', createStepName='"
445         str(step_name) + "', region=a.Set(vertices=loading_points, name='" + str(step_name) + "_1" + str(affix) +
446         "'), c" + str(load_type_letter) + "=1=" + str(force[0]) + ", c" + str(load_type_letter) + "=2=" + str(force[1]) +

```

```

447     ", c" + str(load_type_letter) + "3=" + str(force[2]) +
448     ", distributionType=UNIFORM, field='', localCsys=None)\n")
449 file.close()
450 # ConcentratedForce      f
451 # Moment                  m
452
453
454 # DEFINES BOUNDARY CONDITIONS OF THE MODEL
455 def define_boundary_conditions(filename, step_name, affix, vertices, direction):
456     file = open("".join(filename), "a")
457     condition_type = 'Displacement'
458     file.write("fixes = a.instances['Part-1-1'].vertices.findAt(")
459     for vertex_coordinates in vertices:
460         file.write("(" + str(vertex_coordinates[0]) +
461             ", " + str(vertex_coordinates[1]) +
462             ", " + str(vertex_coordinates[2]) + "), ), ")
463     file.write(")\n")
464     "mdb.models['Model-1'].\" + str(condition_type) + "BC(name='" + str(step_name) + str(affix) +
465     "_BC', createStepName='" + str(step_name) + "', region=a.Set(vertices=fixes, name='" +
466     str(step_name) + "_2" + str(affix) + "'), localCsys=None, " + str(direction) + ")\n")
467 file.close()
468 # condition_types:
469 # Encastre
470 # Pinned
471 # Xsymm
472 # Ysymm
473 # Zsymm
474 # Xasymm
475 # Yasymm
476 # Zasymm
477 # Displacement
478 # direction:
479 # u1, u2, u3, ur1, ur2, ur3
480
481
482 # DEACTIVATES BOUNDARY CONDITIONS OF FORMER STEPS (THEY GET PROPAGATED BY ABAQUS DEFAULT)
483 def deactivate_boundary_conditions(filename, step_name, deactivate_step_name):
484     file = open("".join(filename), "a")
485     file.write("mdb.models['Model-1'].boundaryConditions['" + str(step_name) + "_BC'].deactivate('" +
486     str(deactivate_step_name) + "')\n")
487 file.close()
488
489
490 # DEACTIVATES LOADINGS OF FORMER STEPS (THEY GET PROPAGATED BY ABAQUS DEFAULT)
491 def deactivate_loadings(filename, step_name, deactivate_step_name):
492     file = open("".join(filename), "a")
493     file.write("mdb.models['Model-1'].loads['" + str(step_name) + "_load'].deactivate('" +
494     str(deactivate_step_name) + "')\n")
495 file.close()
496
497
498 # GET THE DISPLACEMENT OF REQUESTED FIELD OUTPUT
499 def request_field_output(filename, step_name, output_name):
500     file = open("".join(filename), "a")
501     file.write("mdb.models['Model-1'].FieldOutputRequest(name='" + str(output_name) + "', createStepName='" +
502     str(step_name) + "', variables=(U,S, ))\n")
503 file.close()
504
505
506 # SUBMITS THE SIMULATION
507 def submit(filename, job_name):
508     file = open("".join(filename), "a")
509     file.write("mdb.Job(name='" + str(job_name) + "', model='Model-1', description='', type=ANALYSIS, "
510     "atTime=None, waitMinutes=0, waitHours=0, queue=None, memory=90, "
511     "memoryUnits=PERCENTAGE, getMemoryFromAnalysis=True, "
512     "explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE, echoPrint=OFF, "
513     "modelPrint=OFF, contactPrint=OFF, historyPrint=OFF, userSubroutine='', scratch='', resultsFormat=ODB)\n"
514     "mdb.jobs['" + str(job_name) + "'].submit(consistencyChecking=OFF)\n"
515     "mdb.jobs['" + str(job_name) + "'].waitForCompletion()\n"
516     ")
517 file.close()
518
519
520 ##########
521 # FUNCTIONS FOR THE OUTPUT EVALUATION OF THE SIMULATION

```

```

522
523 # READ THE DISPLACEMENTS OF ALL SIDES OF A CERTAIN STEP AND RETURNS THEM
524 def read_odb_step(filename, step_name, job_name):
525     file = open("".join(filename), "a")
526     file.write("myOdb = odbAccess.openOdb(path='" + str(job_name) + ".odb')\n")
527     # for step_name in step_name:
528     file.write("Step = myOdb.steps['" + str(step_name) + "']\n"
529             "displacement = Step.frames[1].fieldOutputs['U']\n"
530             "list_of_sides = ['SIGMA_X_1', 'SIGMA_X_2_X', 'SIGMA_Y_1', 'SIGMA_Y_2_Y', 'SIGMA_Z_1', 'SIGMA_Z_2_Z']\n"
531             "node_region=dict()\n"
532             "displacements = dict()\n"
533             "for sides in list_of_sides:\n"
534                 "node_region[sides]=myOdb.rootAssembly.nodeSets[sides]\n"
535                 "values = displacement.getSubset(region=node_region[sides]).values\n"
536                 "displacements[sides]=list()\n"
537                 "for value in values:\n"
538                     "displacements[sides].append(value.data)\n"
539             "results['" + step_name + "'] = displacements\n"
540     file.close()
541
542 #####
543
544
545
546 class Script:
547     # INITIALIZE THE SCRIPT
548     def __init__(self, filename, truss, material, abaqus_path, abaqus_version):
549         self.filename = filename
550         self.truss = truss
551         self.material = material
552
553     # OPEN THE SCRIPT FILE
554     file = open("".join(self.filename), "w")
555     # WRITE INTO FILE
556     file.write("# " + "".join(self.filename) + "\n# This file is automatically generated to run in abaqus\n")
557     import datetime
558     file.write("# This file was generated on " + str(datetime.datetime.now()) +
559             "\n# Max Engensperger, maxe@alumni.ethz.ch\n# Truss name: " + str(truss.name) + "\n# Cell_Size: " +
560             str(truss.cell_size) + "\n# Number of Cells: " + str(truss.number_of_cells) + "\n")
561     file.write(
562             "from abaqus import *\n"
563             "from abaqusConstants import *\n"
564             "import __main__\n"
565             "import section\n"
566             "import regionToolset\n"
567             "import displayGroupMdbToolset as dqm\n"
568             "import part\n"
569             "import material\n"
570             "import assembly\n"
571             "import step\n"
572             "import interaction\n"
573             "import load\n"
574             "import mesh\n"
575             "import optimization\n"
576             "import job\n"
577             "import sketch\n"
578             "import visualization\n"
579             "import xyPlot\n"
580             "import displayGroupOdbToolset as dgo\n"
581             "import connectorBehavior\n"
582             "import sys\n"
583             "import odb\n"
584             "from odbAccess import *\n"
585             "import odbAccess\n"
586             "from odbMaterial import *\n"
587             "from odbSection import *\n"
588             "from abaqusConstants import *\n"
589             "from odbMaterial import *\n"
590             "from odbSection import *\n"
591             "import pickle\n"
592             "sys.path.insert(9, r'" + str(abaqus_path) + str(abaqus_version) +  # c:/SIMULIA/Abaqus
593             "/code/python2.7/lib/abaqus_plugins/stlExport')\n"
594             "import stlExport_kernel\n"
595             "Mdb()\n"
596             "session.viewports['Viewport: 1'].viewportAnnotationOptions.setValues(compassPrivilegedPlane=XYPLANE)\n"

```



```

672                 direction='u2=0')
673     for step in list_of_steps:
674         deactivate_boundary_conditions(self.filename, step_name=step_name + "_X", deactivate_step_name=step)
675         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Y", deactivate_step_name=step)
676         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Z", deactivate_step_name=step)
677         deactivate_loadings(self.filename, step_name=step_name, deactivate_step_name=step)
678         request_field_output(self.filename, step_name=step_name, output_name="OUTPUT_Z")
679         list_of_steps.append(step_name)
680
681     # SIGMA_Y
682
683     step_name = "SIGMA_Y"
684     create_static_step(self.filename, step_name=step_name)
685     vertices = self.truss.find_points_in_plane(axis="y",
686                                                 axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
687                                                 accuracy=accuracy)
688     define_loadings(self.filename, step_name=step_name, affix="", vertices=vertices,
689                     load_type="ConcentratedForce", load_type_letter="f",
690                     force=[0, -applied_force / len(vertices), 0])
691     define_boundary_conditions(self.filename, step_name=step_name, affix="_Y",
692                               vertices=self.truss.find_points_in_plane(axis="y",
693                                               axis_value=-0.5 * self.truss.cell_size,
694                                               accuracy=accuracy),
695                               direction='u2=0')
696     define_boundary_conditions(self.filename, step_name=step_name, affix="_Z",
697                               vertices=self.truss.find_points_in_plane(axis="z",
698                                               axis_value=-0.5 * self.truss.cell_size,
699                                               accuracy=accuracy),
700                               direction='u3=0')
701     define_boundary_conditions(self.filename, step_name=step_name, affix="_X",
702                               vertices=self.truss.find_points_in_plane(axis="x",
703                                               axis_value=-0.5 * self.truss.cell_size,
704                                               accuracy=accuracy),
705                               direction='u1=0')
706     for step in list_of_steps:
707         deactivate_boundary_conditions(self.filename, step_name=step_name + "_X", deactivate_step_name=step)
708         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Y", deactivate_step_name=step)
709         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Z", deactivate_step_name=step)
710         deactivate_loadings(self.filename, step_name=step_name, deactivate_step_name=step)
711         request_field_output(self.filename, step_name=step_name, output_name="OUTPUT_Y")
712         list_of_steps.append(step_name)
713
714     # SIGMA_X
715
716     step_name = "SIGMA_X"
717     create_static_step(self.filename, step_name=step_name)
718     vertices = self.truss.find_points_in_plane(axis="x",
719                                                 axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
720                                                 accuracy=accuracy)
721     define_loadings(self.filename, step_name=step_name, affix="", vertices=vertices,
722                     load_type="ConcentratedForce", load_type_letter="f",
723                     force=[-applied_force / len(vertices), 0, 0])
724     define_boundary_conditions(self.filename, step_name=step_name, affix="_X",
725                               vertices=self.truss.find_points_in_plane(axis="x",
726                                               axis_value=-0.5 * self.truss.cell_size,
727                                               accuracy=accuracy),
728                               direction='u1=0')
729     define_boundary_conditions(self.filename, step_name=step_name, affix="_Y",
730                               vertices=self.truss.find_points_in_plane(axis="y",
731                                               axis_value=-0.5 * self.truss.cell_size,
732                                               accuracy=accuracy),
733                               direction='u2=0')
734     define_boundary_conditions(self.filename, step_name=step_name, affix="_Z",
735                               vertices=self.truss.find_points_in_plane(axis="z",
736                                               axis_value=-0.5 * self.truss.cell_size,
737                                               accuracy=accuracy),
738                               direction='u3=0')
739     for step in list_of_steps:
740         deactivate_boundary_conditions(self.filename, step_name=step_name + "_X", deactivate_step_name=step)
741         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Y", deactivate_step_name=step)
742         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Z", deactivate_step_name=step)
743         deactivate_loadings(self.filename, step_name=step_name, deactivate_step_name=step)
744         request_field_output(self.filename, step_name=step_name, output_name="OUTPUT_X")
745         list_of_steps.append(step_name)
746

```

```

747     # TAU_XZ
748
749     step_name = "TAU_XZ"
750     create_static_step(self.filename, step_name=step_name)
751     vertices = self.truss.find_points_in_plane(axis="z",
752                                                 axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
753                                                 accuracy=accuracy)
754     define_loadings(self.filename, step_name=step_name, affix="_Z", vertices=vertices,
755                     load_type="ConcentratedForce", load_type_letter="f",
756                     force=[applied_force / len(vertices), 0, 0])
757     vertices = self.truss.find_points_in_plane(axis="x",
758                                                 axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
759                                                 accuracy=accuracy)
760     define_loadings(self.filename, step_name=step_name, affix="_X", vertices=vertices,
761                     load_type="ConcentratedForce", load_type_letter="f",
762                     force=[0, 0, applied_force / len(vertices)])
763     define_boundary_conditions(self.filename, step_name=step_name, affix='_Z',
764                               vertices=self.truss.find_points_in_plane(axis="z",
765                                               axis_value=-0.5 * self.truss.cell_size,
766                                               accuracy=accuracy),
767                               direction='u1=0, ur2=0')
768     define_boundary_conditions(self.filename, step_name=step_name, affix='_X',
769                               vertices=self.truss.find_points_in_plane(axis="x",
770                                               axis_value=-0.5 * self.truss.cell_size,
771                                               accuracy=accuracy),
772                               direction='u3=0, ur2=0')
773     define_boundary_conditions(self.filename, step_name=step_name, affix='_Y',
774                               vertices=self.truss.find_points_in_plane(axis="y",
775                                               axis_value=-0.5 * self.truss.cell_size,
776                                               accuracy=accuracy),
777                               direction='u2=0')
778     for step in list_of_steps:
779         deactivate_boundary_conditions(self.filename, step_name=step_name + "_X", deactivate_step_name=step)
780         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Y", deactivate_step_name=step)
781         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Z", deactivate_step_name=step)
782         deactivate_loadings(self.filename, step_name=str(step_name) + "_X", deactivate_step_name=step)
783         deactivate_loadings(self.filename, step_name=str(step_name) + "_Z", deactivate_step_name=step)
784     request_field_output(self.filename, step_name=step_name, output_name="OUTPUT_XZ")
785     list_of_steps.append(step_name)
786
787     # TAU_YZ
788
789     step_name = "TAU_YZ"
790     create_static_step(self.filename, step_name=step_name)
791     vertices = self.truss.find_points_in_plane(axis="z",
792                                                 axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
793                                                 accuracy=accuracy)
794     define_loadings(self.filename, step_name=step_name, affix="_Z", vertices=vertices,
795                     load_type="ConcentratedForce", load_type_letter="f",
796                     force=[0, applied_force / len(vertices), 0])
797     vertices = self.truss.find_points_in_plane(axis="y",
798                                                 axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
799                                                 accuracy=accuracy)
800     define_loadings(self.filename, step_name=step_name, affix="_Y", vertices=vertices,
801                     load_type="ConcentratedForce", load_type_letter="f",
802                     force=[0, 0, applied_force / len(vertices)])
803     define_boundary_conditions(self.filename, step_name=step_name, affix='_Z',
804                               vertices=self.truss.find_points_in_plane(axis="z",
805                                               axis_value=-0.5 * self.truss.cell_size,
806                                               accuracy=accuracy),
807                               direction='u2=0, ur1=0')
808     define_boundary_conditions(self.filename, step_name=step_name, affix='_Y',
809                               vertices=self.truss.find_points_in_plane(axis="y",
810                                               axis_value=-0.5 * self.truss.cell_size,
811                                               accuracy=accuracy),
812                               direction='u3=0, ur1=0')
813     define_boundary_conditions(self.filename, step_name=step_name, affix='_X',
814                               vertices=self.truss.find_points_in_plane(axis="x",
815                                               axis_value=-0.5 * self.truss.cell_size,
816                                               accuracy=accuracy),
817                               direction='u1=0')
818     for step in list_of_steps:
819         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Z", deactivate_step_name=step)
820         deactivate_boundary_conditions(self.filename, step_name=step_name + "_Y", deactivate_step_name=step)
821         deactivate_boundary_conditions(self.filename, step_name=step_name + "_X", deactivate_step_name=step)

```

```

822     deactivate_loadings(self.filename, step_name=str(step_name) + "_Z", deactivate_step_name=step)
823     deactivate_loadings(self.filename, step_name=str(step_name) + "_Y", deactivate_step_name=step)
824     request_field_output(self.filename, step_name=step_name, output_name="OUTPUT_YZ")
825     list_of_steps.append(step_name)
826
827 # TAU_XY
828
829 step_name = "TAU_XY"
830 create_static_step(self.filename, step_name=step_name)
831 vertices = self.truss.find_points_in_plane(axis="y",
832                                         axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
833                                         accuracy=accuracy)
834 define_loadings(self.filename, step_name=step_name, affix="_Y", vertices=vertices,
835                 load_type="ConcentratedForce", load_type_letter="f",
836                 force=[applied_force / len(vertices), 0, 0])
837 vertices = self.truss.find_points_in_plane(axis="x",
838                                         axis_value=(number_of_cells - 0.5) * self.truss.cell_size,
839                                         accuracy=accuracy)
840 define_loadings(self.filename, step_name=step_name, affix="_X", vertices=vertices,
841                 load_type="ConcentratedForce", load_type_letter="f",
842                 force=[0, applied_force / len(vertices), 0])
843 define_boundary_conditions(self.filename, step_name=step_name, affix="_Y",
844                             vertices=self.truss.find_points_in_plane(axis="y",
845                                         axis_value=-0.5 * self.truss.cell_size,
846                                         accuracy=accuracy),
847                             direction='u1=0, ur3=0')
848 define_boundary_conditions(self.filename, step_name=step_name, affix="_X",
849                             vertices=self.truss.find_points_in_plane(axis="x",
850                                         axis_value=-0.5 * self.truss.cell_size,
851                                         accuracy=accuracy),
852                             direction='u2=0, ur3=0')
853 define_boundary_conditions(self.filename, step_name=step_name, affix="_Z",
854                             vertices=self.truss.find_points_in_plane(axis="z",
855                                         axis_value=-0.5 * self.truss.cell_size,
856                                         accuracy=accuracy),
857                             direction='u3=0')
858 for step in list_of_steps:
859     deactivate_boundary_conditions(self.filename, step_name=step_name + "_X", deactivate_step_name=step)
860     deactivate_boundary_conditions(self.filename, step_name=step_name + "_Y", deactivate_step_name=step)
861     deactivate_boundary_conditions(self.filename, step_name=step_name + "_Z", deactivate_step_name=step)
862     deactivate_loadings(self.filename, step_name=str(step_name) + "_Y", deactivate_step_name=step)
863     deactivate_loadings(self.filename, step_name=str(step_name) + "_X", deactivate_step_name=step)
864     request_field_output(self.filename, step_name=step_name, output_name="OUTPUT_XY")
865     list_of_steps.append(step_name)
866
867 if submit_job:
868     submit(self.filename, job_name=self.filename[1])
869 if read_output:
870     read_odb_step(self.filename, 'SIGMA_Z', job_name=self.filename[1])
871     read_odb_step(self.filename, 'SIGMA_Y', job_name=self.filename[1])
872     read_odb_step(self.filename, 'SIGMA_X', job_name=self.filename[1])
873     read_odb_step(self.filename, 'TAU_YZ', job_name=self.filename[1])
874     read_odb_step(self.filename, 'TAU_XZ', job_name=self.filename[1])
875     read_odb_step(self.filename, 'TAU_XY', job_name=self.filename[1])
876
877 # EXTRACTS THE RESULTS AS A DICTIONARY(PYTHON STRUCTURE) IN A BINARY PICKLE FILE
878 def pickle_dump(self):
879
880     file = open("".join(self.filename), "a")
881     file.write("result_file = open('" + self.filename[0] + self.filename[1] + "_results', 'wb')\n"
882                 "pickle.dump(results, result_file)\n"
883                 "result_file.close()\n")
884     file.close()

```

A.4.4 Class_Cell.py

```
1  """
2  This class defines the cell which will be inserted into the truss.
3  The object contains the needed nodes and the connections in between, which are dependent of the nodes to evade errors.
4  """
5
6
7  class Cell:
8      def __init__(self, name, pore_size, nodes, connections, strut_thickesses):
9          self.name = name                      # Name of the cell. F.e. "cube"
10         self.pore_size = pore_size            # Should be determined once per cell-> Maybe determine roundness of pore
11         self.nodes = nodes                  # Syntax: [{"node1x", "node1y", "node1z"}, ...]
12         self.connections = connections     # Syntax: [{"node1", "node2", "thickness"}, ...]
13         self.strut_thickesses = strut_thickesses
```

A.4.5 Class_Truss.py

```

1 """
2 This class defines the truss which will be generated with abaqus script.
3 The object contains the corner nodes for each cell, and the cell that is attached on every node.
4 """
5
6
7 class Truss:
8     def __init__(self, name, nodes, cells, cell_size, number_of_cells):
9         self.name = name
10        # Name of the cells. F.e. "cubetruss"
11        self.nodes = nodes
12        # Syntax: [[["node1x", "node1y", "node1z", "self.cells[0]"], ...]
13        self.cells = cells
14        # Syntax: [{"Cell1", "Cell2", ...}]
15        self.cell_size = cell_size
16        # Unit length of the smallest unit cell
17        self.number_of_cells = number_of_cells
18
19    # FINDS THE POINTS IN A PLANE DEFINED BY THE NORMAL AXIS AND THE COORDINATE WHERE IT INTERSECTS WITH SAID AXIS
20    def find_points_in_plane(self, axis, axis_value, accuracy):
21
22        if axis == "x":
23            array_position = 0
24        if axis == "y":
25            array_position = 1
26        if axis == "z":
27            array_position = 2
28
29        list_of_points = list()
30
31        for truss_nodes in self.nodes:
32            for cell_nodes in truss_nodes[3].nodes:
33                node = [x + y for x, y in zip(truss_nodes, cell_nodes)]
34                if axis_value - accuracy <= node[array_position] <= \
35                    axis_value + accuracy:
36                    check = True
37                    for point in list_of_points:
38                        if abs(node[0] - point[0]) < 1e-3 and \
39                            abs(node[1] - point[1]) < 1e-3 and \
40                            abs(node[2] - point[2]) < 1e-3:
41                            check = False
42                    if check:
43                        list_of_points.append(node)
44
45    # FINDS THE POINTS IN A SPACE DEFINED BY BOUNDARIES FOR EACH OF THE THREE COORDINATE DIRECTIONS
46    def find_points_in_space(self, boundaries):
47
48        list_of_points = list()
49
50        for truss_nodes in self.nodes:
51            for cell_nodes in truss_nodes[3].nodes:
52                node = [x + y for x, y in zip(truss_nodes, cell_nodes)]
53                if boundaries[0][0] <= node[0] <= boundaries[0][1]:
54                    if boundaries[1][0] <= node[1] <= boundaries[1][1]:
55                        if boundaries[2][0] <= node[2] <= boundaries[2][1]:
56                            check = True
57                            for point in list_of_points:
58                                if abs(node[0] - point[0]) < 1e-3 and \
59                                    abs(node[1] - point[1]) < 1e-3 and \
60                                    abs(node[2] - point[2]) < 1e-3:
61                                        check = False
62                            if check:
63                                list_of_points.append(node)
64
65        return list_of_points

```

A.4.6 *library_cell.py*

```

72 [22, 30, strut_thicknesses[0]],
73 [23, 31, strut_thicknesses[0]],
74 [25, 26, strut_thicknesses[0]],
75 [27, 28, strut_thicknesses[0]],
76 [29, 30, strut_thicknesses[0]],
77 [31, 24, strut_thicknesses[0]],
78 [24, 32, strut_thicknesses[0]],
79 [25, 33, strut_thicknesses[0]],
80 [26, 34, strut_thicknesses[0]],
81 [27, 35, strut_thicknesses[0]],
82 [28, 36, strut_thicknesses[0]],
83 [29, 37, strut_thicknesses[0]],
84 [30, 38, strut_thicknesses[0]],
85 [31, 39, strut_thicknesses[0]],
86 [32, 33, strut_thicknesses[0]],
87 [34, 35, strut_thicknesses[0]],
88 [36, 37, strut_thicknesses[0]],
89 [38, 39, strut_thicknesses[0]],
90 [32, 40, strut_thicknesses[0]],
91 [33, 41, strut_thicknesses[0]],
92 [34, 42, strut_thicknesses[0]],
93 [35, 43, strut_thicknesses[0]],
94 [36, 44, strut_thicknesses[0]],
95 [37, 45, strut_thicknesses[0]],
96 [38, 46, strut_thicknesses[0]],
97 [39, 47, strut_thicknesses[0]],
98 [40, 41, strut_thicknesses[0]],
99 [41, 42, strut_thicknesses[0]],
100 [42, 43, strut_thicknesses[0]],
101 [43, 44, strut_thicknesses[0]],
102 [44, 45, strut_thicknesses[0]],
103 [45, 46, strut_thicknesses[0]],
104 [46, 47, strut_thicknesses[0]],
105 [47, 40, strut_thicknesses[0]]]

# PORE SIZE: Describes the pore size of the Cell in standard size 1.
# It is described by the diameter of the biggest fitting sphere in the cell
106 pore_size.append(2 * a)
107 pore_size.append(math.sqrt(3) / 2 * a)
108 pore_size.append((math.sqrt(2) + 1) * a)

109
110
111
112 elif cell_name == "truncated_cube":
113
114     tl = ratio / 2 # standing for truncated_length
115     cell_node_coordinates = [[0, tl, 0], [tl, 0, 0], [1 - tl, 0, 0], [1, tl, 0], [1, 1 - tl, 0], [1 - tl, 1, 0],
116                             [tl, 1, 0], [0, 1 - tl, 0], [0, 0, tl], [1, 0, tl], [1, 1, tl], [0, 1, tl],
117                             [0, 0, 1 - tl], [1, 0, 1 - tl], [0, 1, 1 - tl], [0, tl, 1], [tl, 0, 1],
118                             [1 - tl, 0, 1], [1, tl, 1], [1, 1 - tl, 1], [1 - tl, 1, 1], [tl, 1, 1], [0, 1 - tl, 1]]
119     for x in range(0, len(cell_node_coordinates)):
120         for y in range(0, 3):
121             cell_node_coordinates[x][y] -= 1 / 2

122
123 # CONNECTIONS: Describes the connections via the nodes.
124 node_connections = [[0, 1, strut_thicknesses[0]],
125                      [1, 2, strut_thicknesses[1]],
126                      [2, 3, strut_thicknesses[2]],
127                      [3, 4, strut_thicknesses[3]],
128                      [4, 5, strut_thicknesses[0]],
129                      [5, 6, strut_thicknesses[1]],
130                      [6, 7, strut_thicknesses[2]],
131                      [7, 0, strut_thicknesses[3]],
132                      [0, 8, strut_thicknesses[4]],
133                      [8, 1, strut_thicknesses[5]],
134                      [2, 9, strut_thicknesses[6]],
135                      [9, 3, strut_thicknesses[4]],
136                      [4, 10, strut_thicknesses[7]],
137                      [10, 5, strut_thicknesses[6]],
138                      [6, 11, strut_thicknesses[5]],
139                      [11, 7, strut_thicknesses[7]],
140                      [8, 12, strut_thicknesses[8]],
141                      [9, 13, strut_thicknesses[8]],
142                      [10, 14, strut_thicknesses[8]],
143                      [11, 15, strut_thicknesses[8]],
144                      [12, 16, strut_thicknesses[7]],
145                      [12, 17, strut_thicknesses[6]],
146                      [17, 18, strut_thicknesses[1]],
```

```

147 [18, 13, strut_thicknesses[5]],  

148 [13, 19, strut_thicknesses[7]],  

149 [19, 20, strut_thicknesses[3]],  

150 [20, 14, strut_thicknesses[4]],  

151 [14, 21, strut_thicknesses[5]],  

152 [21, 22, strut_thicknesses[1]],  

153 [22, 15, strut_thicknesses[6]],  

154 [15, 23, strut_thicknesses[4]],  

155 [23, 16, strut_thicknesses[3]],  

156 [16, 17, strut_thicknesses[0]],  

157 [18, 19, strut_thicknesses[2]],  

158 [20, 21, strut_thicknesses[0]],  

159 [22, 23, strut_thicknesses[2]]]  

160 # PORE SIZE: Describes the pore size of the Cell in standard size 1.  

161 # It is described by the diameter of the biggest fitting sphere in the cell  

162 pore_size.append(2 / 3 * math.sqrt(2) * tl)  

163 if 1 - 2 * tl >= math.sqrt(2) * tl:  

164     pore_size.append(1)  

165 else:  

166     pore_size.append(math.sqrt(2) / 2 * (1 - tl))  

167  

168 elif cell_name == "octetrahedron":  

169     cell_node_coordinates = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0], [1 / 2, 1 / 2, 0],  

170         [1 / 2, 0, 1 / 2], [1, 1 / 2, 1 / 2], [1 / 2, 1, 1 / 2], [0, 1 / 2, 1 / 2],  

171         [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1], [1 / 2, 1 / 2, 1], ]  

172     for x in range(0, len(cell_node_coordinates)):  

173         for y in range(0, 3):  

174             cell_node_coordinates[x][y] -= 1 / 2  

175 # CONNECTIONS: Describes the connections via the nodes.  

176 node_connections = [[0, 4, strut_thicknesses[0]],  

177     [1, 4, strut_thicknesses[1]],  

178     [2, 4, strut_thicknesses[0]],  

179     [3, 4, strut_thicknesses[1]],  

180     [0, 5, strut_thicknesses[2]],  

181     [1, 5, strut_thicknesses[3]],  

182     [1, 6, strut_thicknesses[4]],  

183     [2, 6, strut_thicknesses[5]],  

184     [2, 7, strut_thicknesses[3]],  

185     [3, 7, strut_thicknesses[2]],  

186     [3, 8, strut_thicknesses[5]],  

187     [0, 8, strut_thicknesses[4]],  

188     [5, 9, strut_thicknesses[3]],  

189     [5, 10, strut_thicknesses[2]],  

190     [6, 10, strut_thicknesses[5]],  

191     [6, 11, strut_thicknesses[4]],  

192     [7, 11, strut_thicknesses[2]],  

193     [7, 12, strut_thicknesses[3]],  

194     [8, 12, strut_thicknesses[4]],  

195     [8, 9, strut_thicknesses[5]],  

196     [9, 13, strut_thicknesses[0]],  

197     [10, 13, strut_thicknesses[1]],  

198     [11, 13, strut_thicknesses[0]],  

199     [12, 13, strut_thicknesses[1]],  

200     [5, 6, strut_thicknesses[0]],  

201     [6, 7, strut_thicknesses[1]],  

202     [7, 8, strut_thicknesses[0]],  

203     [8, 5, strut_thicknesses[1]],  

204     [4, 5, strut_thicknesses[5]],  

205     [4, 6, strut_thicknesses[2]],  

206     [4, 7, strut_thicknesses[4]],  

207     [4, 8, strut_thicknesses[3]],  

208     [5, 13, strut_thicknesses[4]],  

209     [6, 13, strut_thicknesses[3]],  

210     [7, 13, strut_thicknesses[5]],  

211     [8, 13, strut_thicknesses[2]],  

212     ]  

213 # PORE SIZE: Describes the pore size of the Cell in standard size 1.  

214 # It is described by the diameter of the biggest fitting sphere in the cell  

215 pore_size.append(math.sqrt(2) / 3)  

216  

217 elif cell_name == "void_octetrahedron":  

218     cell_node_coordinates = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0], [1 / 2, 1 / 2, 0],  

219         [1 / 2, 0, 1 / 2], [1, 1 / 2, 1 / 2], [1 / 2, 1, 1 / 2], [0, 1 / 2, 1 / 2],  

220         [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1], [1 / 2, 1 / 2, 1], ]  

221     for x in range(0, len(cell_node_coordinates)):

```

```

222         for y in range(0, 3):
223             cell_node_coordinates[x][y] -= 1 / 2
224     # CONNECTIONS: Describes the connections via the nodes.
225     node_connections = [[0, 4, strut_thicknesses[0]],
226                         [1, 4, strut_thicknesses[1]],
227                         [2, 4, strut_thicknesses[0]],
228                         [3, 4, strut_thicknesses[1]],
229                         [0, 5, strut_thicknesses[2]],
230                         [1, 5, strut_thicknesses[3]],
231                         [1, 6, strut_thicknesses[4]],
232                         [2, 6, strut_thicknesses[5]],
233                         [2, 7, strut_thicknesses[3]],
234                         [3, 7, strut_thicknesses[2]],
235                         [3, 8, strut_thicknesses[5]],
236                         [0, 8, strut_thicknesses[4]],
237                         [5, 9, strut_thicknesses[3]],
238                         [5, 10, strut_thicknesses[2]],
239                         [6, 10, strut_thicknesses[5]],
240                         [6, 11, strut_thicknesses[4]],
241                         [7, 11, strut_thicknesses[2]],
242                         [7, 12, strut_thicknesses[3]],
243                         [8, 12, strut_thicknesses[4]],
244                         [8, 9, strut_thicknesses[5]],
245                         [9, 13, strut_thicknesses[0]],
246                         [10, 13, strut_thicknesses[1]],
247                         [11, 13, strut_thicknesses[0]],
248                         [12, 13, strut_thicknesses[1]]]
249     # PORE SIZE: Describes the pore size of the Cell in standard size 1.
250     # It is described by the diameter of the biggest fitting sphere in the cell
251     pore_size.append(math.sqrt(2) / 3 * 2)
252
253 elif cell_name == "diamond":
254     cell_node_coordinates = [[0, 0, 0], [1 / 2, 1 / 2, 0], [1, 1, 0], [1 / 2, 0, 1 / 2], [1, 1 / 2, 1 / 2],
255                             [1 / 2, 1, 1 / 2], [0, 1 / 2, 1 / 2], [1, 0, 1], [1 / 2, 1 / 2, 1],
256                             [0, 1, 1], [1 / 4, 1 / 4, 1 / 4], [3 / 4, 3 / 4, 1 / 4], [3 / 4, 1 / 4, 3 / 4],
257                             [1 / 4, 3 / 4, 3 / 4], ]
258     for x in range(0, len(cell_node_coordinates)):
259         for y in range(0, 3):
260             cell_node_coordinates[x][y] -= 1 / 2
261     # CONNECTIONS: Describes the connections via the nodes.
262     # node_connections = [
263     #     [0, 10, strut_multiplicator[0]],
264     #     [1, 10, strut_multiplicator[1]],
265     #     [1, 11, strut_multiplicator[2]],
266     #     [2, 11, strut_multiplicator[3]],
267     #     [3, 10, strut_multiplicator[4]],
268     #     [6, 10, strut_multiplicator[5]],
269     #     [4, 11, strut_multiplicator[6]],
270     #     [5, 11, strut_multiplicator[7]],
271     #     [3, 12, strut_multiplicator[8]],
272     #     [4, 12, strut_multiplicator[9]],
273     #     [7, 12, strut_multiplicator[10]],
274     #     [8, 12, strut_multiplicator[11]],
275     #     [5, 13, strut_multiplicator[12]],
276     #     [6, 13, strut_multiplicator[13]],
277     #     [8, 13, strut_multiplicator[14]],
278     #     [9, 13, strut_multiplicator[15]],
279     #     ]
280     node_connections = [
281         [0, 10, strut_thicknesses[0]],
282         [1, 10, strut_thicknesses[1]],
283         [1, 11, strut_thicknesses[0]],
284         [2, 11, strut_thicknesses[1]],
285         [3, 10, strut_thicknesses[2]],
286         [6, 10, strut_thicknesses[3]],
287         [4, 11, strut_thicknesses[2]],
288         [5, 11, strut_thicknesses[3]],
289         [3, 12, strut_thicknesses[0]],
290         [4, 12, strut_thicknesses[1]],
291         [7, 12, strut_thicknesses[2]],
292         [8, 12, strut_thicknesses[3]],
293         [5, 13, strut_thicknesses[1]],
294         [6, 13, strut_thicknesses[0]],
295         [8, 13, strut_thicknesses[2]],
296         [9, 13, strut_thicknesses[3]],
```

```

297     ]
298     # PORE SIZE: Describes the pore size of the Cell in standard size 1.
299     # It is described by the diameter of the biggest fitting sphere in the cell
300     pore_size.append(3 / 8)
301
302     elif cell_name == "cube":
303         # NODES: Describe the nodes in cartesian coordinates.
304         cell_node_coordinates = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0], [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1]]
305         for x in range(0, len(cell_node_coordinates)):
306             for y in range(0, 3):
307                 cell_node_coordinates[x][y] -= 1 / 2
308         # CONNECTIONS: Describes the connections via the nodes.
309         node_connections = [[0, 1, strut_thicknesses[0]],
310                             [1, 2, strut_thicknesses[1]],
311                             [2, 3, strut_thicknesses[0]],
312                             [3, 0, strut_thicknesses[1]],
313                             [0, 4, strut_thicknesses[2]],
314                             [4, 5, strut_thicknesses[0]],
315                             [5, 6, strut_thicknesses[1]],
316                             [6, 7, strut_thicknesses[0]],
317                             [7, 4, strut_thicknesses[1]],
318                             [1, 5, strut_thicknesses[2]],
319                             [2, 6, strut_thicknesses[2]],
320                             [3, 7, strut_thicknesses[2]]]
321
322         # PORE SIZE: Describes the pore size of the Cell in standard size 1.
323         # It is described by the diameter of the biggest fitting sphere in the cell
324         pore_size.append(1)
325
326     elif cell_name == "inv_cube":
327         # NODES: Describe the nodes in cartesian coordinates.
328         cell_node_coordinates = [[1/2, 1/2, 1/2], [1/2, 1/2, 0], [1/2, 0, 1/2], [0, 1/2, 1/2], [1/2, 1, 1/2],
329                                  [1, 1/2, 1/2], [1/2, 1/2, 1]]
330         for x in range(0, len(cell_node_coordinates)):
331             for y in range(0, 3):
332                 cell_node_coordinates[x][y] -= 1 / 2
333         # CONNECTIONS: Describes the connections via the nodes.
334         node_connections = [[0, 1, strut_thicknesses[0]],
335                             [0, 2, strut_thicknesses[1]],
336                             [0, 6, strut_thicknesses[0]],
337                             [0, 4, strut_thicknesses[1]],
338                             [0, 5, strut_thicknesses[2]],
339                             [0, 3, strut_thicknesses[2]]]
340
341         # PORE SIZE: Describes the pore size of the Cell in standard size 1.
342         # It is described by the diameter of the biggest fitting sphere in the cell
343         pore_size.append(1)
344
345     elif cell_name == "pyramid":
346         # NODES: Describe the nodes in cartesian coordinates.
347         p = ratio
348         q = (1 - p) / 2
349         cell_node_coordinates = [[-q, -q, 0], [1 + q, -q, 0], [1 + q, 1 + q, 0], [-q, 1 + q, 0],
350                                  [q, q, 1], [1 - q, q, 1], [1 - q, 1 - q, 1], [q, 1 - q, 1]]
351         for x in range(0, len(cell_node_coordinates)):
352             for y in range(0, 3):
353                 cell_node_coordinates[x][y] -= 1 / 2
354
355         # CONNECTIONS: Describes the connections via the nodes.
356         node_connections = [[0, 1, strut_thicknesses[0]],
357                             [1, 2, strut_thicknesses[1]],
358                             [2, 3, strut_thicknesses[0]],
359                             [3, 0, strut_thicknesses[1]],
360                             [0, 4, strut_thicknesses[2]],
361                             [4, 5, strut_thicknesses[0]],
362                             [5, 6, strut_thicknesses[1]],
363                             [6, 7, strut_thicknesses[0]],
364                             [7, 4, strut_thicknesses[1]],
365                             [1, 5, strut_thicknesses[2]],
366                             [2, 6, strut_thicknesses[2]],
367                             [3, 7, strut_thicknesses[2]]]
368
369         # PORE SIZE: Describes the pore size of the Cell in standard size 1.
370         # It is described by the diameter of the biggest fitting sphere in the cell
371         if (math.sqrt(1 / 4 + (1 - p / 2) ** 2) * math.sin(math.atan(2 / (2 - p))) - math.atan(1 / (2 - p))) >= 1 / 2:
372             pore_size.append(1)

```

```

372     else:
373         pore_size.append(2 * (1 - p / 2) * math.tan(math.atan(1 / (1 - p)) / 2))
374
375     elif cell_name == "pyramid_inv":
376         # NODES: Describe the nodes in cartesian coordinates.
377         p = ratio
378         q = (1 - p) / 2
379         cell_node_coordinates = [[q, q, 0], [1 - q, q, 0], [1 - q, 1 - q, 0], [q, 1 - q, 0],
380                                  [-q, -q, 1], [1 + q, -q, 1], [1 + q, 1 + q, 1], [-q, 1 + q, 1]]
381         for x in range(0, len(cell_node_coordinates)):
382             for y in range(0, 3):
383                 cell_node_coordinates[x][y] -= 1 / 2
384
385         # CONNECTIONS: Describes the connections via the nodes.
386         node_connections = [[0, 1, strut_thicknesses[0]],
387                             [1, 2, strut_thicknesses[1]],
388                             [2, 3, strut_thicknesses[0]],
389                             [3, 0, strut_thicknesses[1]],
390                             [0, 4, strut_thicknesses[2]],
391                             [4, 5, strut_thicknesses[0]],
392                             [5, 6, strut_thicknesses[1]],
393                             [6, 7, strut_thicknesses[0]],
394                             [7, 4, strut_thicknesses[1]],
395                             [1, 5, strut_thicknesses[2]],
396                             [2, 6, strut_thicknesses[2]],
397                             [3, 7, strut_thicknesses[2]]]
398
399         # PORE SIZE: Describes the pore size of the Cell in standard size 1.
400         # It is described by the diameter of the biggest fitting sphere in the cell
401         if (math.sqrt(1 / 4 + (1 - p / 2) ** 2) * math.sin(math.atan(2 / (2 - p)) - math.atan(1 / (2 - p)))) >= 1 / 2:
402             pore_size.append(1)
403         else:
404             pore_size.append(2 * (1 - p / 2) * math.tan(math.atan(1 / (1 - p)) / 2))
405
406     elif cell_name == "pyramid_twist":
407         # NODES: Describe the nodes in cartesian coordinates.
408         p = ratio
409         q = (1 - p) / 2
410         cell_node_coordinates = [[-q, q, 0], [1 + q, q, 0], [1 + q, 1 - q, 0], [-q, 1 - q, 0],
411                                  [q, -q, 1], [1 - q, -q, 1], [1 - q, 1 + q, 1], [q, 1 + q, 1]]
412         for x in range(0, len(cell_node_coordinates)):
413             for y in range(0, 3):
414                 cell_node_coordinates[x][y] -= 1 / 2
415
416         # CONNECTIONS: Describes the connections via the nodes.
417         node_connections = [[0, 1, strut_thicknesses[0]],
418                             [1, 2, strut_thicknesses[1]],
419                             [2, 3, strut_thicknesses[0]],
420                             [3, 0, strut_thicknesses[1]],
421                             [0, 4, strut_thicknesses[2]],
422                             [4, 5, strut_thicknesses[0]],
423                             [5, 6, strut_thicknesses[1]],
424                             [6, 7, strut_thicknesses[0]],
425                             [7, 4, strut_thicknesses[1]],
426                             [1, 5, strut_thicknesses[2]],
427                             [2, 6, strut_thicknesses[2]],
428                             [3, 7, strut_thicknesses[2]]]
429
430         # PORE SIZE: Describes the pore size of the Cell in standard size 1.
431         # It is described by the diameter of the biggest fitting sphere in the cell
432         if (math.sqrt(1 / 4 + (1 - p / 2) ** 2) * math.sin(math.atan(2 / (2 - p)) - math.atan(1 / (2 - p)))) >= 1 / 2:
433             pore_size.append(1)
434         else:
435             pore_size.append(2 * (1 - p / 2) * math.tan(math.atan(1 / (1 - p)) / 2))
436
437     elif cell_name == "pyramid_twist_inv":
438         # NODES: Describe the nodes in cartesian coordinates.
439         p = ratio
440         q = (1 - p) / 2
441         cell_node_coordinates = [[q, -q, 0], [1 - q, -q, 0], [1 - q, 1 + q, 0], [q, 1 + q, 0],
442                                  [-q, q, 1], [1 + q, q, 1], [1 + q, 1 - q, 1], [-q, 1 - q, 1]]
443         for x in range(0, len(cell_node_coordinates)):
444             for y in range(0, 3):
445                 cell_node_coordinates[x][y] -= 1 / 2
446

```

```

447     # CONNECTIONS: Describes the connections via the nodes.
448     node_connections = [[0, 1, strut_thicknesses[0]],
449                          [1, 2, strut_thicknesses[1]],
450                          [2, 3, strut_thicknesses[0]],
451                          [3, 0, strut_thicknesses[1]],
452                          [0, 4, strut_thicknesses[2]],
453                          [4, 5, strut_thicknesses[0]],
454                          [5, 6, strut_thicknesses[1]],
455                          [6, 7, strut_thicknesses[0]],
456                          [7, 4, strut_thicknesses[1]],
457                          [1, 5, strut_thicknesses[2]],
458                          [2, 6, strut_thicknesses[2]],
459                          [3, 7, strut_thicknesses[2]]]
460
461     # PORE SIZE: Describes the pore size of the Cell in standard size 1.
462     # It is described by the diameter of the biggest fitting sphere in the cell
463     if (math.sqrt(1 / 4 + (1 - p / 2) ** 2) * math.sin(math.atan(2 / (2 - p)) - math.atan(1 / (2 - p))) >= 1 / 2:
464         pore_size.append(1)
465     else:
466         pore_size.append(2 * (1 - p / 2) * math.tan(math.atan(1 / (1 - p)) / 2))
467
468 elif cell_name == "face_diagonal_cube":
469     # NODES: Describe the nodes in cartesian coordinates.
470     cell_node_coordinates = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0], [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1]]
471     # strut_multiplicator = 0.2 # unit meters
472     for x in range(0, len(cell_node_coordinates)):
473         for y in range(0, 3):
474             cell_node_coordinates[x][y] -= 1 / 2
475
476     # CONNECTIONS: Describes the connections via the nodes.
477     node_connections = [
478         [0, 1, strut_thicknesses[0]],
479         [1, 2, strut_thicknesses[1]],
480         [2, 3, strut_thicknesses[0]],
481         [3, 0, strut_thicknesses[1]],
482         [0, 4, strut_thicknesses[2]],
483         [4, 5, strut_thicknesses[0]],
484         [5, 6, strut_thicknesses[1]],
485         [6, 7, strut_thicknesses[0]],
486         [7, 4, strut_thicknesses[1]],
487         [1, 5, strut_thicknesses[2]],
488         [2, 6, strut_thicknesses[2]],
489         [3, 7, strut_thicknesses[2]],
490         # [1, 3, strut_multiplicator[3]],
491         # [1, 4, strut_multiplicator[4]],
492         # [1, 6, strut_multiplicator[5]],
493         # [3, 4, strut_multiplicator[6]],
494         # [3, 6, strut_multiplicator[7]],
495         # [4, 6, strut_multiplicator[8]],
496         [1, 3, strut_thicknesses[3]],
497         [1, 4, strut_thicknesses[4]],
498         [1, 6, strut_thicknesses[5]],
499         [3, 4, strut_thicknesses[5]],
500         [3, 6, strut_thicknesses[4]],
501         [4, 6, strut_thicknesses[3]],
502     ]
503
504     # PORE SIZE: Describes the pore size of the Cell in standard size 1.
505     # It is described by the diameter of the biggest fitting sphere in the cell
506     pore_size.append(math.sqrt(2) / 3) # ONLY estimated, needs to be determined
507
508 elif cell_name == "face_diagonal_cube_inv":
509     # NODES: Describe the nodes in cartesian coordinates.
510     cell_node_coordinates = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0], [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1]]
511     # strut_multiplicator = 0.2 # unit meters
512     for x in range(0, len(cell_node_coordinates)):
513         for y in range(0, 3):
514             cell_node_coordinates[x][y] -= 1 / 2
515
516     # CONNECTIONS: Describes the connections via the nodes.
517     node_connections = [
518         [0, 1, strut_thicknesses[0]],
519         [1, 2, strut_thicknesses[1]],
520         [2, 3, strut_thicknesses[0]],
521         [3, 0, strut_thicknesses[1]],
```

```

522     [0, 4, strut_thicknesses[2]],  

523     [4, 5, strut_thicknesses[0]],  

524     [5, 6, strut_thicknesses[1]],  

525     [6, 7, strut_thicknesses[0]],  

526     [6, 7, strut_thicknesses[0]],  

527     [7, 4, strut_thicknesses[1]],  

528     [1, 5, strut_thicknesses[2]],  

529     [2, 6, strut_thicknesses[2]],  

530     [3, 7, strut_thicknesses[2]],  

531     # [0, 2, strut_multiplicator[3]],  

532     # [0, 5, strut_multiplicator[4]],  

533     # [0, 7, strut_multiplicator[5]],  

534     # [2, 5, strut_multiplicator[6]],  

535     # [2, 7, strut_multiplicator[7]],  

536     # [5, 7, strut_multiplicator[8]],  

537     [0, 2, strut_thicknesses[3]],  

538     [0, 5, strut_thicknesses[4]],  

539     [0, 7, strut_thicknesses[5]],  

540     [2, 5, strut_thicknesses[5]],  

541     [2, 7, strut_thicknesses[4]],  

542     [5, 7, strut_thicknesses[3]],  

543  

544
545     # PORE SIZE: Describes the pore size of the Cell in standard size 1.  

546     # It is described by the diameter of the biggest fitting sphere in the cell  

547     pore_size.append(math.sqrt(2) / 3) # ONLY estimated, needs to be determined  

548
549     elif cell_name == "body_centered_cube":  

550         # NODES: Describe the nodes in cartesian coordinates.  

551         cell_node_coordinates = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0], [0, 0, 1],  

552             [1, 0, 1], [1, 1, 1], [0, 1, 1], [1 / 2, 1 / 2, 1 / 2]]  

553         # strut_multiplicator = 0.2 # unit meters  

554         for x in range(0, len(cell_node_coordinates)):  

555             for y in range(0, 3):  

556                 cell_node_coordinates[x][y] -= 1 / 2  

557
558         # CONNECTIONS: Describes the connections via the nodes.  

559         node_connections = [[0, 1, strut_thicknesses[0]],  

560             [1, 2, strut_thicknesses[1]],  

561             [2, 3, strut_thicknesses[0]],  

562             [3, 0, strut_thicknesses[1]],  

563             [0, 4, strut_thicknesses[2]],  

564             [4, 5, strut_thicknesses[0]],  

565             [5, 6, strut_thicknesses[1]],  

566             [6, 7, strut_thicknesses[0]],  

567             [7, 4, strut_thicknesses[1]],  

568             [1, 5, strut_thicknesses[2]],  

569             [2, 6, strut_thicknesses[2]],  

570             [3, 7, strut_thicknesses[2]],  

571             [0, 8, strut_thicknesses[3]],  

572             [1, 8, strut_thicknesses[3]],  

573             [2, 8, strut_thicknesses[3]],  

574             [3, 8, strut_thicknesses[3]],  

575             [4, 8, strut_thicknesses[3]],  

576             [5, 8, strut_thicknesses[3]],  

577             [6, 8, strut_thicknesses[3]],  

578             [7, 8, strut_thicknesses[3]],  

579             # [0, 8, strut_thicknesses[3]],  

580             # [1, 8, strut_thicknesses[4]],  

581             # [2, 8, strut_thicknesses[5]],  

582             # [3, 8, strut_thicknesses[6]],  

583             # [4, 8, strut_thicknesses[5]],  

584             # [5, 8, strut_thicknesses[6]],  

585             # [6, 8, strut_thicknesses[3]],  

586             # [7, 8, strut_thicknesses[4]],  

587         ]  

588
589         # PORE SIZE: Describes the pore size of the Cell in standard size 1.  

590         # It is described by the diameter of the biggest fitting sphere in the cell  

591         pore_size.append = 2/3 * math.sqrt(2) / 2  

592         # ONLY estimated, needs to be determined  

593
594     elif cell_name == "octahedron":  

595         # NODES: Describe the nodes in cartesian coordinates.  

596         cell_node_coordinates = [[1 / 2, 1 / 2, 0], [1 / 2, 0, 1 / 2], [1, 1 / 2, 1 / 2],

```

```

597 [1 / 2, 1, 1 / 2], [0, 1 / 2, 1 / 2], [1 / 2, 1 / 2, 1]
598
599 for x in range(0, len(cell_node_coordinates)):
600     for y in range(0, 3):
601         cell_node_coordinates[x][y] -= 1 / 2
602
603 # CONNECTIONS: Describes the connections via the nodes.
604 node_connections = [[0, 1, strut_thicknesses[0]], [0, 2, strut_thicknesses[1]], [0, 3, strut_thicknesses[2]], [0, 4, strut_thicknesses[3]], [1, 5, strut_thicknesses[2]], [2, 5, strut_thicknesses[3]], [3, 5, strut_thicknesses[0]], [4, 5, strut_thicknesses[1]], [1, 2, strut_thicknesses[4]], [2, 3, strut_thicknesses[5]], [3, 4, strut_thicknesses[4]], [4, 1, strut_thicknesses[5]], ]
605
606
607 # PORE SIZE: Describes the pore size of the Cell in standard size 1.
608 # It is described by the diameter of the biggest fitting sphere in the cell
609 pore_size.append(math.sqrt(2) / 3)
610
611
612 elif cell_name == "truncated_octahedron":
613     # NODES: Describe the nodes in cartesian coordinates.
614     p = ratio / 2
615     cell_node_coordinates = [[1 / 2 - p, 1 / 2, 0], [1 / 2, 1 / 2 - p, 0], [1 / 2 + p, 1 / 2, 0], [1 / 2, 1 / 2 + p, 0], [0, 1 / 2, 1 / 2 - p], [1 / 2, 0, 1 / 2 - p], [1, 1 / 2, 1 / 2 - p], [1 / 2 + p, 1 / 2], [0, 1 / 2 + p, 1 / 2], [0, 1 / 2 - p, 1 / 2], [1 / 2 - p, 0, 1 / 2], [1 / 2 + p, 1 / 2 - p], [1, 1 / 2 - p, 1 / 2], [1, 1 / 2 + p, 1 / 2], [1 / 2 + p, 1, 1 / 2], [1 / 2 - p, 1, 1 / 2], [0, 1 / 2, 1 / 2 + p], [1 / 2, 0, 1 / 2 + p], [1, 1 / 2, 1 / 2 + p], [1 / 2 - p, 1 / 2, 1], [1 / 2, 1 / 2 - p, 1], [1 / 2 + p, 1 / 2, 1], [1 / 2, 1 / 2 + p, 1], [1 / 2 - p, 1 / 2, 1], [1 / 2, 1 / 2 - p, 1], [1 / 2 + p, 1 / 2, 1], [1 / 2, 1 / 2 + p, 1]]
616
617 for x in range(0, len(cell_node_coordinates)):
618     for y in range(0, 3):
619         cell_node_coordinates[x][y] -= 1 / 2
620
621
622 # CONNECTIONS: Describes the connections via the nodes.
623 node_connections = [[0, 1, strut_thicknesses[0]], [1, 2, strut_thicknesses[0]], [2, 3, strut_thicknesses[0]], [0, 3, strut_thicknesses[0]], [0, 4, strut_thicknesses[1]], [1, 5, strut_thicknesses[2]], [2, 6, strut_thicknesses[1]], [3, 7, strut_thicknesses[2]], [4, 8, strut_thicknesses[3]], [4, 9, strut_thicknesses[3]], [5, 10, strut_thicknesses[4]], [6, 11, strut_thicknesses[4]], [6, 12, strut_thicknesses[3]], [6, 13, strut_thicknesses[3]], [7, 14, strut_thicknesses[4]], [7, 15, strut_thicknesses[4]], [8, 16, strut_thicknesses[3]], [9, 16, strut_thicknesses[3]], [10, 17, strut_thicknesses[4]], [11, 17, strut_thicknesses[4]], [12, 18, strut_thicknesses[3]], [13, 18, strut_thicknesses[3]], [14, 19, strut_thicknesses[4]], [15, 19, strut_thicknesses[4]], [16, 20, strut_thicknesses[1]], [17, 21, strut_thicknesses[2]], [18, 22, strut_thicknesses[1]], [19, 23, strut_thicknesses[2]], [20, 21, strut_thicknesses[0]], [21, 22, strut_thicknesses[0]], [22, 23, strut_thicknesses[0]]]

```

```

672 [20, 23, strut_thicknesses[0]],  

673 [9, 10, strut_thicknesses[5]],  

674 [11, 12, strut_thicknesses[5]],  

675 [13, 14, strut_thicknesses[5]],  

676 [15, 8, strut_thicknesses[5]],  

677 ]  

678  

679 # PORE SIZE: Describes the pore size of the Cell in standard size 1.  

680 # It is described by the diameter of the biggest fitting sphere in the cell  

681 pore_size.append(ratio)  

682  

683 elif cell_name == "templar_crosse":  

684     # p = ratio  

685     # p = 0.3 # 0-1  

686     # q = 0.4 # 0-0.5  

687     p = ratio[0]  

688     q = ratio[1] / 2  

689     # NODES: Describe the nodes in cartesian coordinates.  

690     cell_node_coordinates = [[1 / 2 - q, 1 / 2, 0], [1 / 2, 1 / 2 - q, 0], [1 / 2 + q, 1 / 2, 0],  

691         [1 / 2, 1 / 2 + q, 0],  

692         [0, 1 / 2, 1 / 2 - q], [0, 1 / 2 + q, 1 / 2], [0, 1 / 2, 1 / 2 + q],  

693         [0, 1 / 2 - q, 1 / 2],  

694         [1 / 2, 0, 1 / 2 - q], [1 / 2 - q, 0, 1 / 2], [1 / 2, 0, 1 / 2 + q],  

695         [1 / 2 + q, 0, 1 / 2],  

696         [1, 1 / 2, 1 / 2 - q], [1, 1 / 2 - q, 1 / 2], [1, 1 / 2, 1 / 2 + q],  

697         [1, 1 / 2 + q, 1 / 2],  

698         [1 / 2, 1, 1 / 2 - q], [1 / 2 + q, 1, 1 / 2], [1 / 2, 1, 1 / 2 + q],  

699         [1 / 2 - q, 1 / 2, 1], [1 / 2, 1 / 2 - q, 1], [1 / 2 + q, 1 / 2, 1],  

700         [1 / 2, 1 / 2 + q, 1],  

701         [1 / 2 * (1 - p), 1 / 2, 1 / 2 * (1 - p)], [1 / 2, 1 / 2 * (1 - p), 1 / 2 * (1 - p)],  

702         [1 / 2 * (1 + p), 1 / 2, 1 / 2 * (1 - p)],  

703         [1 / 2, 1 / 2 * (1 + p), 1 / 2 * (1 - p)],  

704         [1 / 2 * (1 - p), 1 / 2, 1 / 2 * (1 + p)], [1 / 2, 1 / 2 * (1 - p), 1 / 2 * (1 + p)],  

705         [1 / 2 * (1 + p), 1 / 2, 1 / 2 * (1 + p)],  

706         [1 / 2, 1 / 2 * (1 + p), 1 / 2 * (1 + p)],  

707         [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2], [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2],  

708         [1 / 2 * (1 + p), 1 / 2 * (1 + p), 1 / 2],  

709         [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2]]  

710  

711 for x in range(0, len(cell_node_coordinates)):  

712     for y in range(0, 3):  

713         cell_node_coordinates[x][y] -= 1 / 2  

714  

715     # CONNECTIONS: Describes the connections via the nodes.  

716     node_connections = [[0, 1, strut_thicknesses[0]],  

717         [1, 2, strut_thicknesses[0]],  

718         [2, 3, strut_thicknesses[0]],  

719         [3, 0, strut_thicknesses[0]],  

720         [4, 5, strut_thicknesses[0]],  

721         [5, 6, strut_thicknesses[0]],  

722         [6, 7, strut_thicknesses[0]],  

723         [7, 4, strut_thicknesses[0]],  

724         [8, 9, strut_thicknesses[0]],  

725         [9, 10, strut_thicknesses[0]],  

726         [10, 11, strut_thicknesses[0]],  

727         [11, 8, strut_thicknesses[0]],  

728         [12, 13, strut_thicknesses[0]],  

729         [13, 14, strut_thicknesses[0]],  

730         [14, 15, strut_thicknesses[0]],  

731         [15, 12, strut_thicknesses[0]],  

732         [16, 17, strut_thicknesses[0]],  

733         [17, 18, strut_thicknesses[0]],  

734         [18, 19, strut_thicknesses[0]],  

735         [19, 16, strut_thicknesses[0]],  

736         [20, 21, strut_thicknesses[0]],  

737         [21, 22, strut_thicknesses[0]],  

738         [22, 23, strut_thicknesses[0]],  

739         [23, 20, strut_thicknesses[0]],  

740         [0, 24, strut_thicknesses[0]],  

741         [1, 25, strut_thicknesses[0]],  

742         [2, 26, strut_thicknesses[0]],  

743         [3, 27, strut_thicknesses[0]],  

744         [24, 4, strut_thicknesses[0]],  

745         [25, 8, strut_thicknesses[0]],  

746         [26, 12, strut_thicknesses[0]],  

747

```

```

747 [27, 16, strut_thicknesses[0]],  

748 [6, 28, strut_thicknesses[0]],  

749 [10, 29, strut_thicknesses[0]],  

750 [14, 30, strut_thicknesses[0]],  

751 [18, 31, strut_thicknesses[0]],  

752 [28, 20, strut_thicknesses[0]],  

753 [29, 21, strut_thicknesses[0]],  

754 [30, 22, strut_thicknesses[0]],  

755 [31, 23, strut_thicknesses[0]],  

756 [7, 32, strut_thicknesses[0]],  

757 [32, 9, strut_thicknesses[0]],  

758 [11, 33, strut_thicknesses[0]],  

759 [33, 13, strut_thicknesses[0]],  

760 [15, 34, strut_thicknesses[0]],  

761 [34, 17, strut_thicknesses[0]],  

762 [19, 35, strut_thicknesses[0]],  

763 [35, 5, strut_thicknesses[0]],  

764 ]  

765  

766 # PORE SIZE: Describes the pore size of the Cell in standard size 1.  

767 # It is described by the diameter of the biggest fitting sphere in the cell  

768 pore_size.append(0)  

769  

770 elif cell_name == "templar_alt_crosse":  

771     # p = ratio  

772     # p = 0.3    # 0-1  

773     # q = 0.4    # 0-0.5  

774     # r = 0.05   # 0-0.33  

775     p = ratio[0]  

776     q = ratio[1] / 2  

777     r = ratio[2] / 2  

778     # NODES: Describe the nodes in cartesian coordinates.  

779     cell_node_coordinates = [[1 / 2 - q, 1 / 2, 0], [1 / 2, 1 / 2 - q, 0], [1 / 2 + q, 1 / 2, 0],  

780             [1 / 2, 1 / 2 + q, 0], # 0-3  

781             [0, 1 / 2, 1 / 2 - q], [0, 1 / 2 + q, 1 / 2], [0, 1 / 2, 1 / 2 + q],  

782             [0, 1 / 2 - q, 1 / 2], # 4-7  

783             [1 / 2, 0, 1 / 2 - q], [1 / 2 - q, 0, 1 / 2], [1 / 2, 0, 1 / 2 + q],  

784             [1 / 2 + q, 0, 1 / 2], # 8-11  

785             [1, 1 / 2, 1 / 2 - q], [1, 1 / 2 - q, 1 / 2], [1, 1 / 2, 1 / 2 + q],  

786             [1, 1 / 2 + q, 1 / 2], # 12-15  

787             [1 / 2, 1, 1 / 2 - q], [1 / 2 + q, 1, 1 / 2], [1 / 2, 1, 1 / 2 + q],  

788             [1 / 2 - q, 1, 1 / 2], # 16-19  

789             [1 / 2 - q, 1 / 2, 1], [1 / 2, 1 / 2 - q, 1], [1 / 2 + q, 1 / 2, 1],  

790             [1 / 2, 1 / 2 + q, 1], # 20-23  

791             #####  

792             [1 / 2 * (1 - p), 1 / 2 - r, 1 / 2 * (1 - p)],  

793             [1 / 2 + r, 1 / 2 * (1 - p), 1 / 2 * (1 - p)],  

794             [1 / 2 * (1 + p), 1 / 2 + r, 1 / 2 * (1 - p)],  

795             [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2 * (1 - p)], # 24-27  

796             [1 / 2 * (1 - p), 1 / 2 + r, 1 / 2 * (1 + p)],  

797             [1 / 2 - r, 1 / 2 * (1 - p), 1 / 2 * (1 + p)],  

798             [1 / 2 * (1 + p), 1 / 2 - r, 1 / 2 * (1 + p)],  

799             [1 / 2 + r, 1 / 2 * (1 + p), 1 / 2 * (1 + p)], # 28-31  

800             [1 / 2 * (1 - p), 1 / 2 * (1 - p), 1 / 2 + r],  

801             [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2 + r],  

802             [1 / 2 * (1 + p), 1 / 2 * (1 + p), 1 / 2 + r],  

803             [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2 + r], # 32-35  

804             #####  

805             [1 / 2 * (1 - p), 1 / 2 + r, 1 / 2 * (1 - p)],  

806             [1 / 2 - r, 1 / 2 * (1 - p), 1 / 2 * (1 - p)],  

807             [1 / 2 * (1 + p), 1 / 2 - r, 1 / 2 * (1 - p)],  

808             [1 / 2 + r, 1 / 2 * (1 + p), 1 / 2 * (1 - p)], # 36-39  

809             [1 / 2 * (1 - p), 1 / 2 - r, 1 / 2 * (1 + p)],  

810             [1 / 2 + r, 1 / 2 * (1 - p), 1 / 2 * (1 + p)],  

811             [1 / 2 * (1 + p), 1 / 2 + r, 1 / 2 * (1 + p)],  

812             [1 / 2 - r, 1 / 2 * (1 + p), 1 / 2 * (1 + p)], # 40-43  

813             [1 / 2 * (1 - p), 1 / 2 * (1 - p), 1 / 2 - r],  

814             [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2 - r],  

815             [1 / 2 * (1 + p), 1 / 2 * (1 + p), 1 / 2 - r],  

816             [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2 - r]] # 44-47  

817     #  

818     #####  

819     for x in range(0, len(cell_node_coordinates)):  

820         for y in range(0, 3):  

821             cell_node_coordinates[x][y] -= 1 / 2

```

```

822     # CONNECTIONS: Describes the connections via the nodes.
823     node_connections = [[0, 1, strut_thicknesses[0]],
824                           [1, 2, strut_thicknesses[0]],
825                           [2, 3, strut_thicknesses[0]],
826                           [3, 0, strut_thicknesses[0]],
827                           [4, 5, strut_thicknesses[0]],
828                           [5, 6, strut_thicknesses[0]],
829                           [6, 7, strut_thicknesses[0]],
830                           [7, 4, strut_thicknesses[0]],
831                           [8, 9, strut_thicknesses[0]],
832                           [9, 10, strut_thicknesses[0]],
833                           [10, 11, strut_thicknesses[0]],
834                           [11, 8, strut_thicknesses[0]],
835                           [12, 13, strut_thicknesses[0]],
836                           [13, 14, strut_thicknesses[0]],
837                           [14, 15, strut_thicknesses[0]],
838                           [15, 12, strut_thicknesses[0]],
839                           [16, 17, strut_thicknesses[0]],
840                           [17, 18, strut_thicknesses[0]],
841                           [18, 19, strut_thicknesses[0]],
842                           [19, 16, strut_thicknesses[0]],
843                           [20, 21, strut_thicknesses[0]],
844                           [21, 22, strut_thicknesses[0]],
845                           [22, 23, strut_thicknesses[0]],
846                           [23, 20, strut_thicknesses[0]],
847                           ######[[0, 24, strut_thicknesses[0]],
848                           [0, 24, strut_thicknesses[0]],
849                           [1, 25, strut_thicknesses[0]],
850                           [2, 26, strut_thicknesses[0]],
851                           [3, 27, strut_thicknesses[0]],
852
853                           [24, 36, strut_thicknesses[0]],
854                           [25, 37, strut_thicknesses[0]],
855                           [26, 38, strut_thicknesses[0]],
856                           [27, 39, strut_thicknesses[0]],
857
858                           [36, 4, strut_thicknesses[0]],
859                           [37, 8, strut_thicknesses[0]],
860                           [38, 12, strut_thicknesses[0]],
861                           [39, 16, strut_thicknesses[0]],
862                           ######[[6, 40, strut_thicknesses[0]],
863                           [6, 40, strut_thicknesses[0]],
864                           [10, 41, strut_thicknesses[0]],
865                           [14, 42, strut_thicknesses[0]],
866                           [18, 43, strut_thicknesses[0]],
867
868                           [28, 40, strut_thicknesses[0]],
869                           [29, 41, strut_thicknesses[0]],
870                           [30, 42, strut_thicknesses[0]],
871                           [31, 43, strut_thicknesses[0]],
872
873                           [28, 20, strut_thicknesses[0]],
874                           [29, 21, strut_thicknesses[0]],
875                           [30, 22, strut_thicknesses[0]],
876                           [31, 23, strut_thicknesses[0]],
877                           ######[[7, 44, strut_thicknesses[0]],
878                           [7, 44, strut_thicknesses[0]],
879                           [44, 32, strut_thicknesses[0]],
880                           [32, 9, strut_thicknesses[0]],
881
882                           [11, 45, strut_thicknesses[0]],
883                           [45, 33, strut_thicknesses[0]],
884                           [33, 13, strut_thicknesses[0]],
885
886                           [15, 46, strut_thicknesses[0]],
887                           [46, 34, strut_thicknesses[0]],
888                           [34, 17, strut_thicknesses[0]],
889
890                           [19, 47, strut_thicknesses[0]],
891                           [47, 35, strut_thicknesses[0]],
892                           [35, 5, strut_thicknesses[0]],
893
894
895     # PORE SIZE: Describes the pore size of the Cell in standard size 1.
896     # It is described by the diameter of the biggest fitting sphere in the cell

```

```

897 pore_size.append(0)
898
899 elif cell_name == "templar_alt2_cross":
900     # p = ratio
901     # p = 0.3    # 0-1
902     # q = 0.4    # 0-0.5
903     # r = 0.05   # 0-0.33
904     p = ratio[0]
905     q = ratio[1] / 2
906     r = ratio[2] / 3
907     # NODES: Describe the nodes in cartesian coordinates.
908     cell_node_coordinates = [[1 / 2 - q, 1 / 2 - r, 0], [1 / 2 + r, 1 / 2 - q, 0],
909                               [1 / 2 + q, 1 / 2 + r, 0], [1 / 2 - r, 1 / 2 + q, 0], # 0-3
910                               [0, 1 / 2 + r, 1 / 2 - q], [0, 1 / 2 + q, 1 / 2 + r],
911                               [0, 1 / 2 - r, 1 / 2 + q], [0, 1 / 2 - q, 1 / 2 - r], # 4-7
912                               [1 / 2 - r, 0, 1 / 2 - q], [1 / 2 - q, 0, 1 / 2 + r],
913                               [1 / 2 + r, 0, 1 / 2 + q], [1 / 2 + q, 0, 1 / 2 - r], # 8-11
914                               [1, 1 / 2 - r, 1 / 2 - q], [1, 1 / 2 - q, 1 / 2 + r],
915                               [1, 1 / 2 + r, 1 / 2 + q], [1, 1 / 2 + q, 1 / 2 - r], # 12-15
916                               [1 / 2 + r, 1, 1 / 2 - q], [1 / 2 + q, 1, 1 / 2 + r],
917                               [1 / 2 - r, 1, 1 / 2 + q], [1 / 2 - q, 1, 1 / 2 - r], # 16-19
918                               [1 / 2 - q, 1 / 2 + r, 1], [1 / 2 - r, 1 / 2 - q, 1],
919                               [1 / 2 + q, 1 / 2 - r, 1], [1 / 2 + r, 1 / 2 + q, 1], # 20-23
920
921 ##### Node coordinates for the 20 nodes in the cross-shaped cell #####
922     [1 / 2 * (1 - p), 1 / 2 - r, 1 / 2 * (1 - p)],
923     [1 / 2 + r, 1 / 2 * (1 - p), 1 / 2 * (1 - p)],
924     [1 / 2 * (1 + p), 1 / 2 + r, 1 / 2 * (1 - p)], # 24-27
925     [1 / 2 - r, 1 / 2 * (1 + p), 1 / 2 * (1 - p)],
926     [1 / 2 * (1 - p), 1 / 2 + r, 1 / 2 * (1 + p)],
927     [1 / 2 - r, 1 / 2 * (1 - p), 1 / 2 * (1 + p)],
928     [1 / 2 * (1 + p), 1 / 2 - r, 1 / 2 * (1 + p)], # 28-31
929     [1 / 2 * (1 - p), 1 / 2 * (1 - p), 1 / 2 + r],
930     [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2 + r],
931     [1 / 2 * (1 + p), 1 / 2 * (1 + p), 1 / 2 + r],
932     [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2 + r], # 32-35
933
934 ##### Node coordinates for the 16 nodes in the top rectangular cell #####
935     [1 / 2 * (1 - p), 1 / 2 + r, 1 / 2 * (1 - p)],
936     [1 / 2 - r, 1 / 2 * (1 - p), 1 / 2 * (1 - p)],
937     [1 / 2 * (1 + p), 1 / 2 - r, 1 / 2 * (1 - p)], # 36-39
938     [1 / 2 + r, 1 / 2 * (1 + p), 1 / 2 * (1 - p)],
939     [1 / 2 + r, 1 / 2 * (1 - p), 1 / 2 * (1 + p)],
940     [1 / 2 * (1 + p), 1 / 2 + r, 1 / 2 * (1 + p)],
941     [1 / 2 - r, 1 / 2 * (1 + p), 1 / 2 * (1 + p)], # 40-43
942     [1 / 2 * (1 - p), 1 / 2 * (1 - p), 1 / 2 - r],
943     [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2 - r],
944     [1 / 2 * (1 + p), 1 / 2 * (1 + p), 1 / 2 - r],
945     [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2 - r]] # 44-47
946
947     #
948     for x in range(0, len(cell_node_coordinates)):
949         for y in range(0, 3):
950             cell_node_coordinates[x][y] -= 1 / 2
951
952     # CONNECTIONS: Describes the connections via the nodes.
953     node_connections = [[0, 1, strut_thicknesses[0]],
954                           [1, 2, strut_thicknesses[0]],
955                           [2, 3, strut_thicknesses[0]],
956                           [3, 0, strut_thicknesses[0]],
957                           [4, 5, strut_thicknesses[0]],
958                           [5, 6, strut_thicknesses[0]],
959                           [6, 7, strut_thicknesses[0]],
960                           [7, 4, strut_thicknesses[0]],
961                           [8, 9, strut_thicknesses[0]],
962                           [9, 10, strut_thicknesses[0]],
963                           [10, 11, strut_thicknesses[0]],
964                           [11, 8, strut_thicknesses[0]],
965                           [12, 13, strut_thicknesses[0]],
966                           [13, 14, strut_thicknesses[0]],
967                           [14, 15, strut_thicknesses[0]],
968                           [15, 12, strut_thicknesses[0]],
969                           [16, 17, strut_thicknesses[0]],
970                           [17, 18, strut_thicknesses[0]],
971                           [18, 19, strut_thicknesses[0]],
972                           [19, 16, strut_thicknesses[0]],
973
974     
```

```

972 [20, 21, strut_thicknesses[0]],
973 [21, 22, strut_thicknesses[0]],
974 [22, 23, strut_thicknesses[0]],
975 [23, 20, strut_thicknesses[0]],
976 ##########
977 [0, 24, strut_thicknesses[0]],
978 [1, 25, strut_thicknesses[0]],
979 [2, 26, strut_thicknesses[0]],
980 [3, 27, strut_thicknesses[0]],
981
982 [24, 36, strut_thicknesses[0]],
983 [25, 37, strut_thicknesses[0]],
984 [26, 38, strut_thicknesses[0]],
985 [27, 39, strut_thicknesses[0]],
986
987 [36, 4, strut_thicknesses[0]],
988 [37, 8, strut_thicknesses[0]],
989 [38, 12, strut_thicknesses[0]],
990 [39, 16, strut_thicknesses[0]],
991 ##########
992 [6, 40, strut_thicknesses[0]],
993 [10, 41, strut_thicknesses[0]],
994 [14, 42, strut_thicknesses[0]],
995 [18, 43, strut_thicknesses[0]],
996
997 [28, 40, strut_thicknesses[0]],
998 [29, 41, strut_thicknesses[0]],
999 [30, 42, strut_thicknesses[0]],
1000 [31, 43, strut_thicknesses[0]],
1001
1002 [28, 20, strut_thicknesses[0]],
1003 [29, 21, strut_thicknesses[0]],
1004 [30, 22, strut_thicknesses[0]],
1005 [31, 23, strut_thicknesses[0]],
1006 ##########
1007 [7, 44, strut_thicknesses[0]],
1008 [44, 32, strut_thicknesses[0]],
1009 [32, 9, strut_thicknesses[0]],
1010
1011 [11, 45, strut_thicknesses[0]],
1012 [45, 33, strut_thicknesses[0]],
1013 [33, 13, strut_thicknesses[0]],
1014
1015 [15, 46, strut_thicknesses[0]],
1016 [46, 34, strut_thicknesses[0]],
1017 [34, 17, strut_thicknesses[0]],
1018
1019 [19, 47, strut_thicknesses[0]],
1020 [47, 35, strut_thicknesses[0]],
1021 [35, 5, strut_thicknesses[0]],
1022 ]
1023
1024 # PORE SIZE: Describes the pore size of the Cell in standard size 1.
1025 # It is described by the diameter of the biggest fitting sphere in the cell
1026 pore_size.append(0)
1027
1028 elif cell_name == "templar_alt2_cross_inv":
1029     # p = ratio
1030     # p = 0.3    # 0-1
1031     # q = 0.4    # 0-0.5
1032     # r = 0.05   # 0-0.33
1033     p = ratio[0]
1034     q = ratio[1] / 2
1035     r = ratio[2] / 3
1036     # NODES: Describe the nodes in cartesian coordinates.
1037     cell_node_coordinates = [[1 / 2 - q, 1 / 2 - r, 0], [1 / 2 + r, 1 / 2 - q, 0],
1038                               [1 / 2 + q, 1 / 2 + r, 0], [1 / 2 - r, 1 / 2 + q, 0], # 0-3
1039                               [0, 1 / 2 + r, 1 / 2 - q], [0, 1 / 2 + q, 1 / 2 + r],
1040                               [0, 1 / 2 - r, 1 / 2 + q], [0, 1 / 2 - q, 1 / 2 - r], # 4-7
1041                               [1 / 2 - r, 0, 1 / 2 - q], [1 / 2 - q, 0, 1 / 2 + r],
1042                               [1 / 2 + r, 0, 1 / 2 + q], [1 / 2 + q, 0, 1 / 2 - r], # 8-11
1043                               [1, 1 / 2 - r, 1 / 2 - q], [1, 1 / 2 - q, 1 / 2 + r],
1044                               [1, 1 / 2 + r, 1 / 2 + q], [1, 1 / 2 + q, 1 / 2 - r], # 12-15
1045                               [1 / 2 + r, 1, 1 / 2 - q], [1 / 2 + q, 1, 1 / 2 + r],
1046                               [1 / 2 - r, 1, 1 / 2 + q], [1 / 2 - q, 1, 1 / 2 - r], # 16-19

```

```

1047 [1 / 2 - q, 1 / 2 + r, 1], [1 / 2 - r, 1 / 2 - q, 1],
1048 [1 / 2 + q, 1 / 2 - r, 1], [1 / 2 + r, 1 / 2 + q, 1], # 20-23
1049 ##########
1050 [1 / 2 * (1 - p), 1 / 2 - r, 1 / 2 * (1 - p)],
1051 [1 / 2 + r, 1 / 2 * (1 - p), 1 / 2 * (1 - p)],
1052 [1 / 2 * (1 + p), 1 / 2 + r, 1 / 2 * (1 - p)],
1053 [1 / 2 - r, 1 / 2 * (1 + p), 1 / 2 * (1 - p)], # 24-27
1054 [1 / 2 * (1 - p), 1 / 2 + r, 1 / 2 * (1 + p)],
1055 [1 / 2 - r, 1 / 2 * (1 - p), 1 / 2 * (1 + p)],
1056 [1 / 2 * (1 + p), 1 / 2 - r, 1 / 2 * (1 + p)],
1057 [1 / 2 + r, 1 / 2 * (1 + p), 1 / 2 * (1 + p)], # 28-31
1058 [1 / 2 * (1 - p), 1 / 2 * (1 - p), 1 / 2 + r],
1059 [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2 + r],
1060 [1 / 2 * (1 + p), 1 / 2 * (1 + p), 1 / 2 + r],
1061 [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2 + r], # 32-35
1062 #####
1063 [1 / 2 * (1 - p), 1 / 2 + r, 1 / 2 * (1 - p)],
1064 [1 / 2 - r, 1 / 2 * (1 - p), 1 / 2 * (1 - p)],
1065 [1 / 2 * (1 + p), 1 / 2 - r, 1 / 2 * (1 - p)],
1066 [1 / 2 + r, 1 / 2 * (1 + p), 1 / 2 * (1 - p)], # 36-39
1067 [1 / 2 * (1 - p), 1 / 2 - r, 1 / 2 * (1 + p)],
1068 [1 / 2 + r, 1 / 2 * (1 - p), 1 / 2 * (1 + p)],
1069 [1 / 2 * (1 + p), 1 / 2 + r, 1 / 2 * (1 + p)],
1070 [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2 * (1 + p)], # 40-43
1071 [1 / 2 * (1 - p), 1 / 2 * (1 - p), 1 / 2 - r],
1072 [1 / 2 * (1 + p), 1 / 2 * (1 - p), 1 / 2 - r],
1073 [1 / 2 * (1 + p), 1 / 2 * (1 + p), 1 / 2 - r],
1074 [1 / 2 * (1 - p), 1 / 2 * (1 + p), 1 / 2 - r]] # 44-47
1075 #
1076 ##########
1077 for x in range(0, len(cell_node_coordinates)):
1078     for y in range(0, 3):
1079         cell_node_coordinates[x][y] -= 1 / 2
1080         cell_node_coordinates[x][2] = -cell_node_coordinates[x][2]
1081
# CONNECTIONS: Describes the connections via the nodes.
1082 node_connections = [[0, 1, strut_thicknesses[0]],
1083                     [1, 2, strut_thicknesses[0]],
1084                     [2, 3, strut_thicknesses[0]],
1085                     [3, 0, strut_thicknesses[0]],
1086                     [4, 5, strut_thicknesses[0]],
1087                     [5, 6, strut_thicknesses[0]],
1088                     [6, 7, strut_thicknesses[0]],
1089                     [7, 4, strut_thicknesses[0]],
1090                     [8, 9, strut_thicknesses[0]],
1091                     [9, 10, strut_thicknesses[0]],
1092                     [10, 11, strut_thicknesses[0]],
1093                     [11, 8, strut_thicknesses[0]],
1094                     [12, 13, strut_thicknesses[0]],
1095                     [13, 14, strut_thicknesses[0]],
1096                     [14, 15, strut_thicknesses[0]],
1097                     [15, 12, strut_thicknesses[0]],
1098                     [16, 17, strut_thicknesses[0]],
1099                     [17, 18, strut_thicknesses[0]],
1100                     [18, 19, strut_thicknesses[0]],
1101                     [19, 16, strut_thicknesses[0]],
1102                     [20, 21, strut_thicknesses[0]],
1103                     [21, 22, strut_thicknesses[0]],
1104                     [22, 23, strut_thicknesses[0]],
1105                     [23, 20, strut_thicknesses[0]],
1106 #####
1107                     [0, 24, strut_thicknesses[0]],
1108                     [1, 25, strut_thicknesses[0]],
1109                     [2, 26, strut_thicknesses[0]],
1110                     [3, 27, strut_thicknesses[0]],
1111
1112                     [24, 36, strut_thicknesses[0]],
1113                     [25, 37, strut_thicknesses[0]],
1114                     [26, 38, strut_thicknesses[0]],
1115                     [27, 39, strut_thicknesses[0]],
1116
1117                     [36, 4, strut_thicknesses[0]],
1118                     [37, 8, strut_thicknesses[0]],
1119                     [38, 12, strut_thicknesses[0]],
1120                     [39, 16, strut_thicknesses[0]],
1121 #####

```

```

1122 [6, 40, strut_thicknesses[0]],
1123 [10, 41, strut_thicknesses[0]],
1124 [14, 42, strut_thicknesses[0]],
1125 [18, 43, strut_thicknesses[0]],
1126
1127 [28, 40, strut_thicknesses[0]],
1128 [29, 41, strut_thicknesses[0]],
1129 [30, 42, strut_thicknesses[0]],
1130 [31, 43, strut_thicknesses[0]],
1131
1132 [28, 20, strut_thicknesses[0]],
1133 [29, 21, strut_thicknesses[0]],
1134 [30, 22, strut_thicknesses[0]],
1135 [31, 23, strut_thicknesses[0]],
1136 ##########
1137 [7, 44, strut_thicknesses[0]],
1138 [44, 32, strut_thicknesses[0]],
1139 [32, 9, strut_thicknesses[0]],
1140
1141 [11, 45, strut_thicknesses[0]],
1142 [45, 33, strut_thicknesses[0]],
1143 [33, 13, strut_thicknesses[0]],
1144
1145 [15, 46, strut_thicknesses[0]],
1146 [46, 34, strut_thicknesses[0]],
1147 [34, 17, strut_thicknesses[0]],
1148
1149 [19, 47, strut_thicknesses[0]],
1150 [47, 35, strut_thicknesses[0]],
1151 [35, 5, strut_thicknesses[0]],
1152 ]
1153
1154 # PORE SIZE: Describes the pore size of the Cell in standard size 1.
1155 # It is described by the diameter of the biggest fitting sphere in the cell
1156 pore_size.append(0)
1157 elif cell_name == "tetrocta": # These are basically just octahedrons cut on a different plane. A.k.a.
1158 # NODES: Describe the nodes in cartesian coordinates.
1159 cell_node_coordinates = [[1 / 2, 0, 0], [1, 1 / 2, 0], [1 / 2, 1, 0], [0, 1 / 2, 0],
1160 [0, 0, 1 / 2], [1, 0, 1 / 2], [1, 1, 1 / 2], [0, 1, 1 / 2],
1161 [1 / 2, 0, 1], [1, 1 / 2, 1], [1 / 2, 1, 1], [0, 1 / 2, 1],
1162 [1 / 2, 1 / 2, 1 / 2]]
1163 for x in range(0, len(cell_node_coordinates)):
1164     for y in range(0, 3):
1165         cell_node_coordinates[x][y] -= 1 / 2
1166 # CONNECTIONS: Describes the connections via the nodes.
1167 node_connections = [[0, 1, strut_thicknesses[0]],
1168 [1, 2, strut_thicknesses[1]],
1169 [2, 3, strut_thicknesses[2]],
1170 [3, 0, strut_thicknesses[3]],
1171 [8, 9, strut_thicknesses[0]],
1172 [9, 10, strut_thicknesses[1]],
1173 [10, 11, strut_thicknesses[2]],
1174 [11, 8, strut_thicknesses[3]],
1175 [0, 5, strut_thicknesses[4]],
1176 [1, 5, strut_thicknesses[5]],
1177 [1, 6, strut_thicknesses[6]],
1178 [2, 6, strut_thicknesses[7]],
1179 [2, 7, strut_thicknesses[8]],
1180 [3, 7, strut_thicknesses[9]],
1181 [3, 4, strut_thicknesses[10]],
1182 [0, 4, strut_thicknesses[11]],
1183 [4, 8, strut_thicknesses[12]],
1184 [5, 8, strut_thicknesses[13]],
1185 [5, 9, strut_thicknesses[14]],
1186 [6, 9, strut_thicknesses[15]],
1187 [6, 10, strut_thicknesses[16]],
1188 [7, 10, strut_thicknesses[17]],
1189 [7, 11, strut_thicknesses[18]],
1190 [4, 11, strut_thicknesses[19]]]
1191
1192 # PORE SIZE: Describes the pore size of the Cell in standard size 1.
1193 # It is described by the diameter of the biggest fitting sphere in the cell
1194 pore_size.append(1)
1195
1196 elif cell_name == "file_super_truss":

```

```

1197     # NODES: Describe the nodes in cartesian coordinates.
1198     cell_node_coordinates = csv_reader.csv_read_nodes("C:\\\\Users\\\\Maxe-PC2\\\\Desktop\\\\code_and_csv\\\\"
1199                                                 "1nC[10]tL[1]tG[1]uL[1000]di[200]/nodes.csv", 0)
1200     # for x in range(0, len(cell_node_coordinates)):
1201     #     for y in range(0, 3):
1202     #         cell_node_coordinates[x][y] -= 1/2
1203
1204     # CONNECTIONS: Describes the connections via the nodes.
1205     node_connections = csv_reader.csv_read_mems("C:\\\\Users\\\\Maxe-PC2\\\\Desktop\\\\code_and_csv\\\\"
1206                                                 "1nC[10]tL[1]tG[1]uL[1000]di[200]/mems.csv", 0)
1207
1208     # PORE SIZE: Describes the pore size of the Cell in standard size 1.
1209     # It is described by the diameter of the biggest fitting sphere in the cell
1210     pore_size.append(0)
1211
1212 elif cell_name == "square":
1213     # NODES: Describe the nodes in cartesian coordinates.
1214     cell_node_coordinates = [[0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0]]
1215     # strut_multiplicator = 0.2 # unit meters
1216     for x in range(0, len(cell_node_coordinates)):
1217         for y in range(0, 2):
1218             cell_node_coordinates[x][y] -= 1 / 2
1219
1220     # CONNECTIONS: Describes the connections via the nodes.
1221     node_connections = [[0, 1],
1222                         [1, 2],
1223                         [2, 3],
1224                         [3, 0]]
1225
1226     # PORE SIZE: Describes the pore size of the Cell in standard size 1.
1227     # It is described by the diameter of the biggest fitting sphere in the cell
1228     pore_size.append(0)
1229
1230 else:
1231     node_connections = [None, None]
1232     cell_node_coordinates = None
1233     pore_size = None
1234     exit("ERROR: cell_name specified is not defined in the library")
1235
# DEFINE the strut thickness of each strut
1236
1237 if len(node_connections[1]) < 3:
1238     counter = 0
1239     for struts in node_connections:
1240         struts.append(strut_thicknesses[counter])
1241         counter += 1
1242     del counter
1243
1244 # SIZING change to cell size
1245 for x in range(0, len(cell_node_coordinates)):
1246     for y in range(0, 3):
1247         cell_node_coordinates[x][y] *= cell_size
1248
for counter in range(0, len(pore_size)):
1249     pore_size[counter] *= cell_size
1250     pore_size[counter] -= strut_thicknesses[0]
1251     if pore_size[counter] < 0:
1252         pore_size[counter] = 0
1253
# GENERATE beginning and ending coordinates of each connection in cartesian coordinates.
1254 cell_node_connections = list()
1255 for connection_points in node_connections:
1256     cell_node_connections.append([cell_node_coordinates[connection_points[0]],
1257                                 cell_node_coordinates[connection_points[1]],
1258                                 connection_points[2]])
1259
1260 cell = Cell(name=(cell_name + affix), pore_size=pore_size,
1261             nodes=cell_node_coordinates, connections=cell_node_connections, strut_thicknesses=strut_thicknesses)
1262
return cell

```

A.4.7 library_truss.py

```

1 """
2 This file creates and returns the nodes of a truss and defines the containing cells.
3 """
4
5 from Class_Truss import Truss
6 from library_cell import generate_cell
7
8 def generate_truss(truss_name, affix, cell_size, strut_thicknesses, number_of_cells, cell_ratio):
9     # NODES: Describe the nodes and their affiliated cells in cartesian coordinates.
10    cells = list()
11    nodes = list()
12    if (truss_name == "cubes" or truss_name == "inv_cubes" or truss_name == "body_centered_cubes" or
13        truss_name == "octahedrons" or
14        truss_name == "truncated_cubes" or truss_name == "diamonds" or truss_name == "varying_truncated_cubes" or
15        truss_name == "face_diagonal_cubes" or truss_name == "octetrahedrons" or truss_name == "void_octetrahedrons" or
16        truss_name == "templar_crosses" or truss_name == "templar_alt_crosses" or truss_name == "tetroctas" or
17        truss_name == "truncated_octahedrons"):
18        cells.append(generate_cell(cell_name=truss_name[:len(truss_name) - 1], affix="",
19                                  strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
20        for x in range(0, number_of_cells):
21            for y in range(0, number_of_cells):
22                for z in range(0, number_of_cells):
23                    nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[0]])
24    elif truss_name == "face_diagonal_cubes_alt":
25        if number_of_cells == 1:
26            cells.append(generate_cell(cell_name="face_diagonal_cube", affix="",
27                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
28            nodes.append([0, 0, 0, cells[0]])
29        else:
30            cells.append(generate_cell(cell_name="face_diagonal_cube", affix="",
31                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
32            cells.append(generate_cell(cell_name="face_diagonal_cube_inv", affix="",
33                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
34
35        for x in range(0, number_of_cells):
36            for y in range(0, number_of_cells):
37                for z in range(0, number_of_cells):
38                    if (x + y + z) % 2 == 0:
39                        nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[0]])
40                    else:
41                        nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[1]])
42    elif truss_name == "templar_alt2_crosses":
43        cells.append(generate_cell(cell_name="templar_alt2_cross", affix="",
44                                  strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
45        cells.append(generate_cell(cell_name="templar_alt2_cross_inv", affix="",
46                                  strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
47
48        for x in range(0, number_of_cells):
49            for y in range(0, number_of_cells):
50                for z in range(0, number_of_cells):
51                    if (x + y + z) % 2 == 0:
52                        nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[0]])
53                    else:
54                        nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[1]])
55    elif truss_name == "pyramids":
56        if number_of_cells == 1:
57            cells.append(generate_cell(cell_name="pyramid", affix="",
58                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
59            nodes.append([0, 0, 0, cells[0]])
60        else:
61            cells.append(generate_cell(cell_name="pyramid", affix="",
62                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
63            cells.append(generate_cell(cell_name="pyramid_inv", affix="",
64                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
65            cells.append(generate_cell(cell_name="pyramid_twist", affix="",
66                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
67            cells.append(generate_cell(cell_name="pyramid_twist_inv", affix="",
68                                      strut_thicknesses=strut_thicknesses, cell_size=cell_size, ratio=cell_ratio))
69
70        for x in range(0, number_of_cells):
71            for y in range(0, number_of_cells):
72                for z in range(0, number_of_cells):

```

```

72     if z % 2 == 0:
73         if y % 2 == 0:
74             if x % 2 == 0:
75                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[0]])
76             else:
77                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[3]])
78         else:
79             if x % 2 == 0:
80                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[2]])
81             else:
82                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[1]])
83     else:
84         if y % 2 == 0:
85             if x % 2 == 0:
86                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[1]])
87             else:
88                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[2]])
89         else:
90             if x % 2 == 0:
91                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[3]])
92             else:
93                 nodes.append([x * cell_size, y * cell_size, z * cell_size, cells[0]])
94 elif truss_name == "file_super_truss":
95     cells.append(generate_cell(cell_name="file_super_truss", affix="",
96                               strut_thickesses=[], cell_size=1, ratio=cell_ratio))
97     nodes.append([0, 0, 0, cells[0]])
98
99 else:
100     print("Possible Names are: \n"
101           "cubes\nbody_centered_cubes\noctahedrons\ntruncated_cubes\npyramids\ndiamonds\nvarying_truncated_cubes\n"
102           "face_diagonal_cubes\nface_diagonal_cubes_alt\noctetrahedrons\nvoid_octetrahedrons\ntemplar_crosses\n"
103           "templar_alt_crosses\nfile_super_truss\ntruncated_octahedrons\ntetraoctas")
104     nodes = None
105     cells = None
106     exit("ERROR: truss_name specified is not defined in the library")
107
108 truss = Truss(name=(truss_name + affix),
109                nodes=nodes,
110                cells=cells,
111                cell_size=cell_size,
112                number_of_cells=number_of_cells)
113
return truss

```

A.4.8 *csv_reader.py*

```

1 import csv
2
3
4 def csv_read_mems(path, multiplicator):
5     connections = [[]]
6     connections.pop(0)
7     i = 0
8     # C:\\Users / Maxe - PC2 / Desktop / code_and_csv / mems.csv
9     with open(path, newline='') as csvfile:
10         conn_reader = csv.reader(csvfile, delimiter=',')
11         for row in conn_reader:
12             i += 1
13             # row2 = [map(int, x) for x in row[2]]
14             # if(i>2): connections.append([row2, row[3], row[13]])
15             # row2 = [map(int, x) for x in row[2]]
16             # if(i>2): connections.append([list(map(int, row[2])), row[3], row[13]])
17             if i > 2:
18                 connections.append([int(row[2]) - 1, int(row[3]) - 1, float(row[13]) * 1e-6])
19             # if(i>2): connections.append([list(map(int, row[2])), row[3], row[13]])
20             # print(row[0]+"\t"+row[2]+"\t"+row[3])
21     return connections
22
23
24 def csv_read_nodes(path, multiplicator):
25     coordinates = [[]]
26     coordinates.pop(0)
27     i = 0
28     # "C:\\\\Users\\\\Maxe-PC2\\\\Desktop\\\\code_and_csv\\\\nodes.csv"
29     with open(path, newline='') as csvfile:
30         coord_reader = csv.reader(csvfile, delimiter=',')
31         for row in coord_reader:
32             i += 1
33             if i > 2:
34                 coordinates.append([float(row[3]) * 1e-3, float(row[4]) * 1e-3, float(row[5]) * 1e-3])
35
36     return coordinates
37
38
39 def csv_read_list(path):
40
41     values = [[]]
42     values.pop(0)
43     i = 0
44     # "C:\\\\Users\\\\Maxe-PC2\\\\Desktop\\\\code_and_csv\\\\nodes.csv"
45     with open(path, newline='') as csvfile:
46         coord_reader = csv.reader(csvfile, delimiter=',')
47         for row in coord_reader:
48             value_row = list()
49             for value in row:
50                 value_row.append(float(value))
51             values.append(value_row)
52
53
54     return values

```

BIBLIOGRAPHY

- [1] X. Wang, J.s. Nyman, X. Dong, H. Leng, and M. Reyes. Fundamental biomechanics in bone tissue engineering. *Synthesis Lectures on Tissue Engineering*, 2(1):1–225, 2010. doi: 10.2200/s00246ed1v01y200912tis004.
- [2] Robert E. Guldberg and Angel O. Duty. Design parameters for engineering bone regeneration. *Functional Tissue Engineering*, page 146–161. doi: 10.1007/0-387-21547-6_12.
- [3] J. C. Reichert and D. W. Hutmacher. Bone tissue engineering. *Tissue Engineering*, page 431–456, 2010. doi: 10.1007/978-3-642-02824-3_21.
- [4] Susmita Bose, Sahar Vahabzadeh, and Amit Bandyopadhyay. Bone tissue engineering using 3d printing. *Materials Today*, 16(12):496–504, 2013. doi: 10.1016/j.mattod.2013.11.017.
- [5] M.i.z. Ridzwan ., Solehuddin Shuib ., A.y. Hassan ., A.a. Shokri ., and M.n. Mohamad Ibrahim . Problem of stress shielding and improvement to the hip implant designs: A review. *Journal of Medical Sciences(Faisalabad) J. of Medical Sciences*, 7(3):460–467, Jan 2007. doi: 10.3923/jms.2007.460.467.
- [6] F Chen, X Feng, W Wu, H Ouyang, Z Gao, X Cheng, R Hou, and T Mao. Segmental bone tissue engineering by seeding osteoblast precursor cells into titanium mesh–coral composite scaffolds. *International journal of oral and maxillofacial surgery*, 36(9):822–827, 2007.
- [7] Jan-Thorsten Schantz, Dietmar Werner Hutmacher, Christopher Xu Fu Lam, Maik Brinkmann, Kit Mui Wong, Thiam Chye Lim, Ning Chou, Robert Erling Guldberg, and Swee Hin Teoh. Repair of calvarial defects with customised tissue-engineered bone grafts ii. evaluation of cellular efficiency and efficacy in vivo. *Tissue engineering*, 9(4, Supplement 1): 127–139, 2003.
- [8] Mary S Tyler and David P McCobb. The genesis of membrane bone in the embryonic chick maxilla: epithelial-mesenchymal tissue recombination studies. *Development*, 56(1):269–281, 1980.
- [9] M.c. Kruyt, S.m. Van Gaalen, F.c. Oner, A.j. Verbout, J.d. De Bruijn, and W.j.a. Dhert. Bone tissue engineering and spinal fusion: the potential of hybrid constructs by combining osteoprogenitor cells and scaffolds. *Biomaterials*, 25(9):1463–1473, 2004. doi: 10.1016/s0142-9612(03)00490-3.

- [10] Karen J.l Burg, Scott Porter, and James F Kellam. Biomaterial developments for bone tissue engineering. *Biomaterials*, 21(23):2347–2359, 2000. doi: 10.1016/s0142-9612(00)00102-2.
- [11] V Karageorgiou and D Kaplan. Porosity of 3d biomaterial scaffolds and osteogenesis. *Biomaterials*, 26(27):5474–5491, 2005. doi: 10.1016/j.biomaterials.2005.02.002.
- [12] Marc Bohner. Resorbable biomaterials as bone graft substitutes. *Materials Today*, 13(1-2):24–30, 2010. doi: 10.1016/s1369-7021(10)70014-6.
- [13] Paul A. Banaszkiewicz. Porous-coated hip replacement. the factors governing bone ingrowth, stress shielding, and clinical results. *Classic Papers in Orthopaedics*, page 51–55, Jul 2013. doi: 10.1007/978-1-4471-5451-8_12.
- [14] Mark P. Staiger, Alexis M. Pietak, Jerawala Huadmai, and George Dias. Magnesium and its alloys as orthopedic biomaterials: A review. *Biomaterials*, 27(9):1728–1734, 2006. doi: 10.1016/j.biomaterials.2005.10.003.
- [15] C. E. Wen, Y. Yamada, K. Shimojima, Y. Chino, H. Hosokawa, and M. Mabuchi. Novel titanium foam for bone tissue engineering. *Journal of Materials Research J. Mater. Res.*, 17(10):2633–2639, 2002. doi: 10.1557/jmr.2002.0382.
- [16] John C Middleton and Arthur J Tipton. Synthetic biodegradable polymers as orthopedic devices. *Biomaterials*, 21(23):2335–2346, 2000. doi: 10.1016/s0142-9612(00)00101-0.
- [17] Jessica M. Williams, Adebisi Adewunmi, Rachel M. Schek, Colleen L. Flanagan, Paul H. Krebsbach, Stephen E. Feinberg, Scott J. Hollister, and Suman Das. Bone tissue engineering using polycaprolactone scaffolds fabricated via selective laser sintering. *Biomaterials*, 26(23):4817–4827, 2005. doi: 10.1016/j.biomaterials.2004.11.057.
- [18] K. Rezwan, Q.z. Chen, J.j. Blaker, and Aldo Roberto Boccaccini. Biodegradable and bioactive porous polymer/inorganic composite scaffolds for bone tissue engineering. *Biomaterials*, 27(18):3413–3431, 2006. doi: 10.1016/j.biomaterials.2006.01.039.
- [19] Malcolm N. Cooke, John P. Fisher, David Dean, Clare Rimnac, and Antonios G. Mikos. Use of stereolithography to manufacture critical-sized 3d biodegradable scaffolds for bone ingrowth. *Journal of Biomedical Materials Research J. Biomed. Mater. Res.*, 64B(2):65–69, Jun 2003. doi: 10.1002/jbm.b.10485.
- [20] Qizhi Chen, Chenghao Zhu, and George A Thouas. Progress and challenges in biomaterials used for bone tissue engineering: bioactive glasses and elastomeric composites. *Prog Biomater Progress in Biomaterials*, 1(1):2, 2012. doi: 10.1186/2194-0517-1-2.

- [21] S. S. Liao, F. Z. Cui, W. Zhang, and Q. L. Feng. Hierarchically biomimetic bone scaffold materials: Nano-ha/collagen/pla composite. *Journal of Biomedical Materials Research J. Biomed. Mater. Res.*, 69B(2):158–165, 2004. doi: 10.1002/jbm.b.20035.
- [22] Wanpeng Cao and Larry L. Hench. Bioactive materials. *Ceramics International*, 22(6):493–507, 1996. doi: 10.1016/0272-8842(95)00126-3.
- [23] The virtual institute for artificial electromagnetic materials and metamaterials. <http://www.metamorphose-vi.org/index.php/metamaterials>. Accessed: September 2nd, 2016.
- [24] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer, 2006.
- [25] Liliana Polo-Corrales, Magda Latorre-Esteves, and Jaime E. Ramirez-Vick. Scaffold design for bone regeneration. *Journal of Nanoscience and Nanotechnology j. nanosci. nanotech.*, 14(1):15–56, Jan 2014. doi: 10.1166/jnn.2014.9127.
- [26] X. Zheng, H. Lee, T. H. Weisgraber, M. Shusteff, J. Deotte, E. B. Duoss, J. D. Kuntz, M. M. Biener, Q. Ge, J. A. Jackson, and et al. Ultralight, ultrastiff mechanical metamaterials. *Science*, 344(6190):1373–1377, 2014. doi: 10.1126/science.1252291.
- [27] Martin Philip Bendsøe and Noboru Kikuchi. Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering*, 71(2):197–224, 1988. doi: 10.1016/0045-7825(88)90086-2.
- [28] B. S. Bucklen, W.a. Wettergreen, E. Yuksel, and M.a.k. Liebschner. Bone-derived cad library for assembly of scaffolds in computer-aided tissue engineering. *Virtual and Physical Prototyping*, 3(1):13–23, 2008. doi: 10.1080/17452750801911352.
- [29] Andy L. Olivares, Èlia Marsal, Josep A. Planell, and Damien Lacroix. Finite element study of scaffold architecture design and culture conditions for tissue engineering. *Biomaterials*, 30(30):6142–6149, 2009. doi: 10.1016/j.biomaterials.2009.07.041.
- [30] Ferry P.w. Melchels, Katia Bertoldi, Ruggero Gabbrielli, Aldrik H. Velders, Jan Feijen, and Dirk W. Grijpma. Mathematically defined tissue engineering scaffold architectures prepared by stereolithography. *Biomaterials*, 31(27):6909–6916, 2010. doi: 10.1016/j.biomaterials.2010.05.068.
- [31] L. E. Murr, S. M. Gaytan, F. Medina, H. Lopez, E. Martinez, B. I. Machado, D. H. Hernandez, L. Martinez, M. I. Lopez, R. B. Wicker, and et al. Next-generation biomedical implants using additive manufacturing of complex, cellular and functional mesh arrays. *Philosophical Transactions of the Royal Society London A*, 370(1962):20120362, 2012. doi: 10.1098/rsta.2012.0362.

- Royal Society A: Mathematical, Physical and Engineering Sciences*, 368(1917): 1999–2032, 2010. doi: [10.1098/rsta.2010.0010](https://doi.org/10.1098/rsta.2010.0010).
- [32] Jan Wieding, Andreas Wolf, and Rainer Bader. Numerical optimization of open-porous bone scaffold structures to match the elastic properties of human cortical bone. *Journal of the Mechanical Behavior of Biomedical Materials*, 37:56–68, 2014. doi: [10.1016/j.jmbbm.2014.05.002](https://doi.org/10.1016/j.jmbbm.2014.05.002).
 - [33] S Rajagopalan and R Robb. Schwarz meets schwann: Design and fabrication of biomorphic and durataxic tissue engineering scaffolds. *Medical Image Analysis*, 10(5):693–712, 2006. doi: [10.1016/j.media.2006.06.001](https://doi.org/10.1016/j.media.2006.06.001).
 - [34] Zhongzhong Chen, Xilan Feng, and Zhiqiang Jiang. Structural design and optimization of interior scaffolds in artificial bone. *2007 1st International Conference on Bioinformatics and Biomedical Engineering*, 2007. doi: [10.1109/icbbe.2007.88](https://doi.org/10.1109/icbbe.2007.88).
 - [35] M.f Ashby. The properties of foams and lattices. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 364 (1838):15–30, 2006. doi: [10.1098/rsta.2005.1678](https://doi.org/10.1098/rsta.2005.1678).
 - [36] Sebastian C. Kapfer, Stephen T. Hyde, Klaus Mecke, Christoph H. Arns, and Gerd E. Schröder-Turk. Minimal surface scaffold designs for tissue engineering. *Biomaterials*, 32(29):6875–6882, 2011. doi: [10.1016/j.biomaterials.2011.06.012](https://doi.org/10.1016/j.biomaterials.2011.06.012).
 - [37] W. Sun, B. Starly, J. Nam, and A. Darling. Bio-cad modeling and its applications in computer-aided tissue engineering. *Computer-Aided Design*, 37(11):1097–1114, 2005. doi: [10.1016/j.cad.2005.02.002](https://doi.org/10.1016/j.cad.2005.02.002).
 - [38] Qing-Xi Hu, Yi Qian, Yuan Yao, Jun Guo, and Yongwei Yu. A computer designing approach for generation of controllable porosity in the bone scaffold. *2009 IEEE 10th International Conference on Computer-Aided Industrial Design and Conceptual Design*, 2009. doi: [10.1109/caidcd.2009.5375116](https://doi.org/10.1109/caidcd.2009.5375116).
 - [39] Aldo R Boccaccini and Jonny J Blaker. Bioactive composite materials for tissue engineering scaffolds. *Expert Review of Medical Devices*, 2(3): 303–317, 2005. doi: [10.1586/17434440.2.3.303](https://doi.org/10.1586/17434440.2.3.303).
 - [40] Dietmar W. Hutmacher. Scaffolds in tissue engineering bone and cartilage. *Biomaterials*, 21(24):2529–2543, 2000. doi: [10.1016/s0142-9612\(00\)00121-6](https://doi.org/10.1016/s0142-9612(00)00121-6).
 - [41] Dong-Jin Yoo. Computer-aided porous scaffold design for tissue engineering using triply periodic minimal surfaces. *International Journal of Precision Engineering and Manufacturing Int. J. Precis. Eng. Manuf.*, 12(1): 61–71, 2011. doi: [10.1007/s12541-011-0008-9](https://doi.org/10.1007/s12541-011-0008-9).

- [42] Y. Shikinami, K. Okazaki, M. Saito, M. Okuno, S. Hasegawa, J. Tamura, S. Fujibayashi, and T. Nakamura. Bioactive and bioresorbable cellular cubic-composite scaffolds for use in bone reconstruction. *Journal of The Royal Society Interface*, 3(11):805–821, 2006. doi: 10.1098/rsif.2006.0144.

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):

Vorname(n):

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „[Zitier-Knigge](#)“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum

Unterschrift(en)

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.