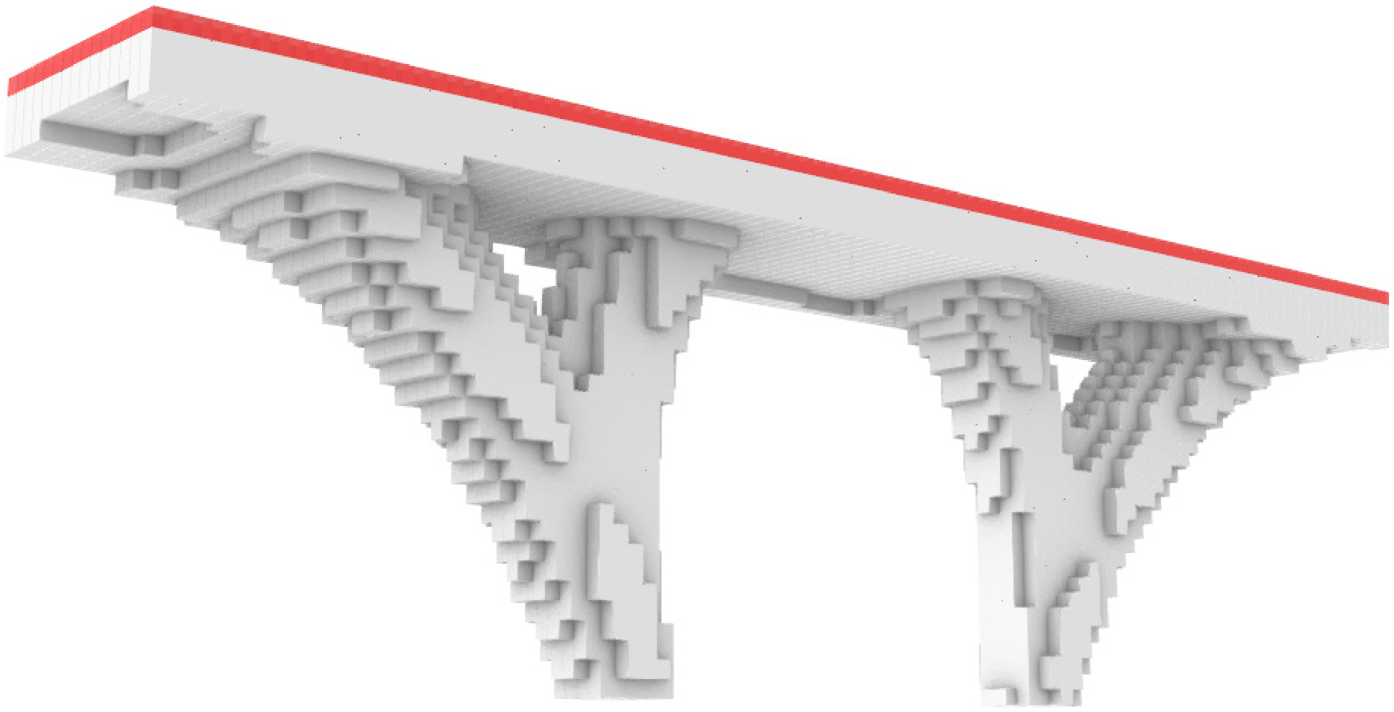
Topology optimization as architectural form finding



Rick van Dijk *4373618*
Master's thesis 02/06/20

Topology optimization as architectural form finding

Master's thesis by Rick van Dijk | 4373618
02/07/2020

Delft University of Technology

Faculty of Architecture and the Built Environment

Working with new materials requires new methods on how geometry can be shaped. This research is a step towards the holy grail for computational designers; an algorithm that fully designs a building. From shape to bricklaying patterns. The chair of design informatics gave me the chance to contribute to this holy grail by researching the subject of Topology Optimization.

This mathematical process is mostly used in mechanical engineering and aerospace engineering. The hypothesis of this research is that architecture could benefit from this process as well, mostly because it can also be used in the 3D space. Giving the computer a few inputs and retrieving a shape that is structurally strong and satisfy the constraints of the design. That is the goal of this research.

I hope this research shows the possibilities of Topology Optimization in architecture and explains to readers how the process takes shape. Opening the 'black box' and looking what happends in each step. I think understanding how a calculation is performed, is essential in using it properly and up until its full potential.

I want to thank Pirouz Nourian for his enthousiasm in the subject, in the trust that he put in me and his clear remarks on my work. I want to thank Matthijs Langelaar for taking the time to explain most subjects where he has a profound knowledge. I learnt a huge amount and still am deeply interested in the subject. If possible I would love to spend a few extra years on the topic.

I hope this research finds you interested,

Rick van Dijk

I n d e x

01 Introduction

This first chapter will describe the research framework that was created to guide the research. It provides an overview of the whole research by introducing the problem, the objective and the research questions to answer. It will cover the boundaries of this research, the chosen scope and the motivations behind most of the choices. Lastly it will cover the methodology of the research and the planning that comes along with the chosen methodology.

Figure 1: *Earth as material*

>>>1.1 Background and motivation

Building more sustainable is the primary challenge within Building Technology and has many different ways of pursuing the goal of more sustainable buildings. When looking into these different approaches most vary a lot, from the design of a building to very complicated details. One thing these approaches have in common, is the way that the building is considered. To build sustainable, one has to consider the full life of a building, from the cost of materials and the function of the building to the waste and/or possibilities to re-use the building (John et al., 2005).

On a material level, one sustainable approach is that of building with clay as a material. With new techniques, complex shapes can be generated to work very efficient with this form of masonry. As clay can be reduced to earth and water, the material cost and waste is very low, while the material can last for a long time (Wienerberger, 2020). Considering masonry, the techniques used to create the shapes are essential for masonry to work as efficient as it does. Therefore, sustainable building is not only a new way of designing, but more a integration of architecture and different engineering approaches, such as electrical, mechanical and structural engineering (John et al., 2005).

When focusing on these techniques, topology optimization is a technique commonly used in mechanical and structural engineering to create complex geometry. The goal in this process is to generate a high stiffness, while pursuing the lowest amount of volume. While this sounds promising, the technique is very little used in architecture, while this might be a very efficient way to develop designs.

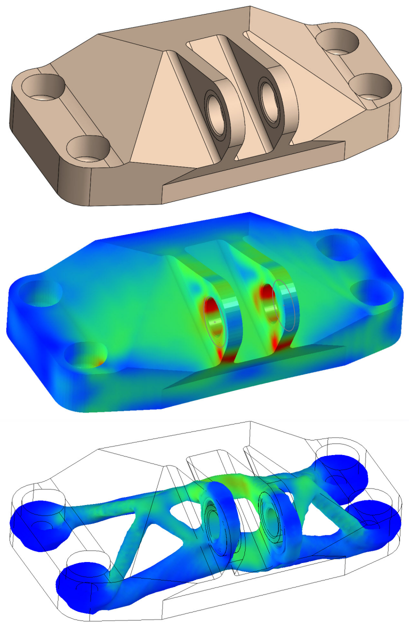


Figure 2: *Example of topology optimization in a hinge*

>>>1.2 Problem statement

The previous chapter already spoke of the possibility of integrating topology optimization in architecture, this chapter will further define the problem and the scope that this research will be in. As previously said, topology optimization is an efficient way to generate shapes that maximize the stiffness, while trying to reduce the volume. Currently there are very little implementations of topology optimization in architecture, while this is quite promising. Architecture follows most of the conceptual boundaries that topology needs, while it also needs a lot of adjusting.

The main idea behind topology optimization is to calculate what voxels (or pixels) in an element are important for the stiffness and what voxels can be removed. Because there is no preconceived shape, topology optimization can create innovative and high-performance shapes (Liu & Tovar, 2014). Usually topology optimization is performed on small objects like beams, hinges or connection pieces. When looking at architecture, the result should be a big structure, which requires a different approach on how to apply topology optimization.

When looking at a building, the forces working are given by the force that lies on a floor and the weight of the structure itself, self-weight. As Bruyneel and Duysinx wrote in their research, it is not easy to apply self-weight to topology optimization, because the density will influence multiple variables (Bruyneel & Duysinx, 2005). Placing an element will increase the stiffness, but the self-weight can cause the structure to result in a lower stiffness. So, to apply topology optimization, self-weight has to be applied in order to give meaningful results. Other important additions that need to be made are found in a roofconstraint, area loads and snowloads.

There are several topology optimization and FEM software programs available, but most are not open source or accessible for students. The result of this research should be accessable for everyone and usable in computational processes. It is preferred to only use open-source programs or languages, like Python. Another problem that arises often in topology optimization is the calculation time. As a lot of calculations have to performed per iteration, calculations usually are long. Focusing on calculation time will also enlarge the usability of the code.

As the main audience for this research consists of students in the field of Architecture, the aim for this research is also to be as explicit as possible when explaining the maths behind the process. It can be assummed that most people reading this research, have little understanding of FEA and similar problems as this one.

In conclusion the problem is written as *"Topology optimization is an often-used method to generate complex shapes with a maximized stiffness and little volume. Implementations in (masonry) architecture look promising, but require the implementation of force dependent loads"*

>>>1.3 Research objective

The main objective of this research is to *implement self-weight in topology optimization, and apply this algorithm to buildings*. When applying topology optimization to buildings, interesting insights might be given on how to build with sustainable materials. This is a step into applying topology optimization further in architecture, which might lead to material saving while having strong structural properties. As the focus is on masonry, it is assumed that when applying self-weight on the structure, the structure will be compression-only. Ideally, this should result in an algorithm, without the use of commercial software, that is easy accessible for architects and students. One big factor in this is the calculation time, which will decrease the possibilities a lot.

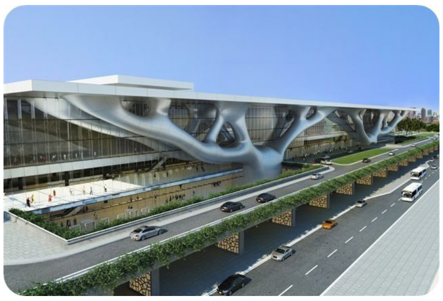


Figure 3: *QNCC, one of the few applications of TO in architecture*

To achieve this objective, first small sub-objectives are created that will lead into a successful result. To implement self-weight in the topology optimization process, first the methodology behind topology optimization has to be understood. Therefore the first sub-objective is *to create a working topology optimization methodology and translate this in the form of an algorithm.*

When this is understood and working, the implementation of self-weight can be made. This requires a strong understanding of the mathematical methodology to be properly used. Secondly a roof constraint and snowloads are essential to solving the system without initial forces. Therefore the second objective is *to implement architectural adjustments in the methodology and in the algorithm.*

Lastly, to retrieve any results that can be used in architecture, the translation to the 3D space has to be made. When the methodology is done precise, the translation can be easily made. It can be seen that a proper methodology in the first sub-objective is key to achieving the other sub-objectives. The final sub-objective of this research is to *generalize the methodology and apply the algorithm to the 3D design spaces.*

The focus of this research is mainly towards applying topology optimization in architecture. Topology optimization comes in many different variants, it can work with forces and heat transfer, but also on hydraulic networks (Bathe, 2006). In this research, the focus will only be on structural topology optimization. Therefore, whenever in this research "topology optimization" or "TO" is mentioned, structural topology optimization is assumed. When applying topology optimization to architecture, many different approaces can be taken. This research will follow previously performed research by Ivan Avdic and courses at the TU Delft, which focusses on masonry buildings. As the scope will be limited to masonry buildings, topology optimization without adjustments will not suffice when applied. The self-weight of the structure and the fact that masonry must be compression-only can not be ignored and have to be implemented to give meaningful results.

Some interesting approaches will not be looked at, while they offer interesting insights to topology optimization in architecture. For instance the use of multi-materials (Huang & Xie, 2009), the placement of components (Zhang et al., 2012) and the optimal layout for topology optimization will not be looked at. Also, this research will go in depth on the methodology behind topology optimization and will not focus on the implementation in the design methodology.

This research will be performed mostly in Python, as this is easy to implement in different software packages, especially focusing on Grasshopper. When looked at architecture, two case studies will be used from a graph-theory level and it is looked at what topology optimization could mean for the design of a building. Also, the several toy problems that are introduced are fictional, but the constraints are based on a real life situation. In these toy problems several simplifications are made, but these will be discussed later. Lastly, the algorithm is going to take a long time to calculate, so optimization of the algorithm should be looked at. However, having a proper result is more important than calculation time.

>>>1.4 Research questions

This research follows the very broad, almost philosophical, question: *"How can topology optimization be implemented in architecture?"*. As this research will not be able to answer this question, the focus has been placed on masonry buildings and the methodological challenges that follow. Therefore the main research question is:

"How can we design structures for masonry buildings using topology optimization?"

From this main research questions several sub-questions arise:

"How does topology optimization work, both in mathematical methodology and in programming?"

"How can topology optimization be utilized in designing in masonry buildings?"

"How can topology optimization transform the design process of masonry buildings, particularly their configuring and shaping processes?"

Many other questions can be asked towards the application of topology optimization in architecture, the question if it is even a viable approach arises for instance. These questions, and other questions will mainly be discussed in the reflection.

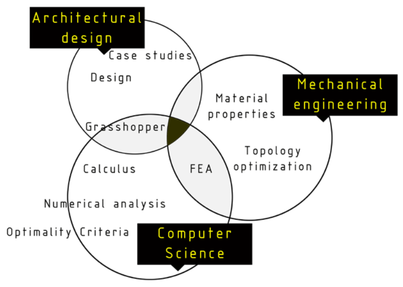


Figure 4: *Euler diagram of this research*

>>>1.5 Research scope and limits

This research combines three scientific disciplines in an interesting way. Architectural design, mechanical engineering and computer science are an excellent intersection for new and innovative research. Shown in the diagram of figure 4, the scope of this research is an integration between parametric design, numerical analysis and material behaviour. In this research, the focus will first be mainly about the combination of mechanical engineering and computer science, followed by the parametric translation. Lastly the focus will be put on the implementation in the field of architecture.

As this research is based from the architectural field, the audience is assumed to be architecture focussed. Concepts from computer science and structural research will therefore be more broadly explained. In contradiction to many architectural research, this research will not contain a main design. The main objective is to create a methodology and algorithm for architectural purposes. Testing this computationally and the use of TOY problems will be the main method of pursuing this.

While this thesis relates to the following subjects in some ways, they fall out of the scope for this research:

- Derivation and modifications of the stiffness matrix
- Verified calculations with SI-units
- Layout optimization
- Architectural design methodology

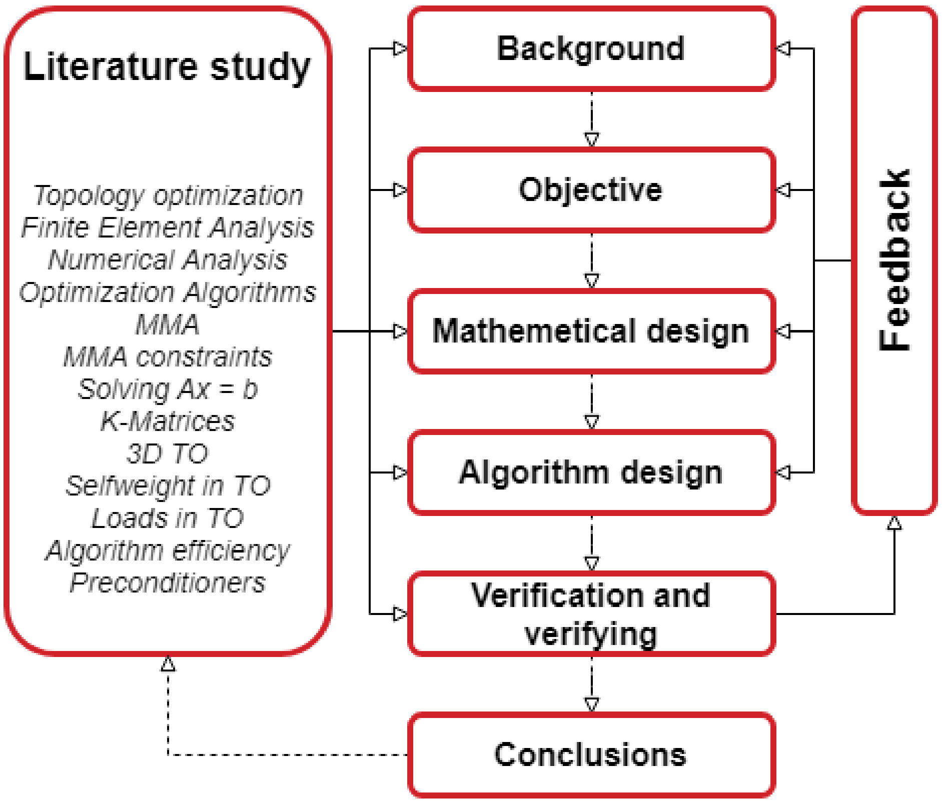


Figure 5: *Methodology of this research*

>>>1.6 Methodology

This chapter will explain the proposed methodology in answering the main research question while firstly solving the sub-goals. It will also discuss main challenges and approaches on how to solve these. A visual representation of the proposed methodology can be seen in figure 5.

First it is important to note that this research is not an exact research and the deliverable will not contain any validated data. However, this research will focus on designing a methodology to implement topology optimization in architecture. As there is a distinctive objective in this research, the main process will cover the following steps, as described by (Peffers et al., 2007).

- Problem identification and motivation
- Objective of the solution
- Design and development
- Demonstration
- Evaluation
- Communication

This research will follow these steps, where previously already the problem identification and motivation are described. To achieve the solution, the objective has to be stated in combination with the proposed methodology. As communication is the scientific publication, the rest of the process can be summarized as (Peffers et al., 2007):

- Build
- Evaluate
- Theorize
- Justify

In this research, the build-phase exists of two different steps, namely building the mathematical model, and the design of the algorithm. As these steps are quite different in approach, they will be handled separately. The next step contains the verification and validation of the design and providing feedback for the previous steps. This feedback should be implemented in the previous steps, up until a level where the design is properly validated and conclusions can be drawn. The process is also represented in figure 5.

>>>1.7 Proposed design methodology

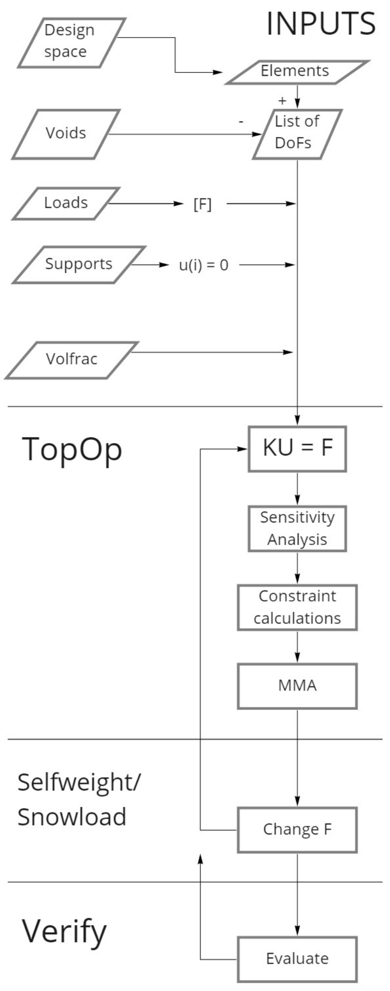


***Figure 6**: Design methodology of this research*

The proposed design methodology, as shown in figure 6, is divided in four phases. The first phase is the retrieval of the inputs, the topology optimization procedure, including FEA and MMA, the application of selfweight and snowloads and then the evaluation. Figure 6 is the basis of the algorithm that is going to be written in Python and is the main objective of this research.

Initializing the inputs is the start to make proper calculations in the design space. Figure 6 shows these inputs. First, the design space is created as a large volume in which the algorithm has the freedom to place or remove mass. For numbering purposes, this is usually assumed to be a cube with integer sizes. Assuming a voxel size of 1 unit, the cube will be initialized with the amount of voxels that are in the cube. Each voxel is called an element with 8 nodes on itself and each node has three degrees of freedoms (or two in 2D problems). Keeping a proper administration of these node IDs and the DoF IDs is very important in the algorithm. The voids are the second input, which are pieces of the cube in no mass can be placed. Lastly, loads and supports are given as input and translated to the force vector [F] and displacement vector [U]. To make it easy for final users, making an inituitive environment is very important.

Secondly the inputs are translated into the topology optimization procedure, proposed by Bensoe and Sigmund (1995). As this procedure is already known, the main challenge will be the translation to the 3D field. A broad understanding of this procedure will ensure that the translation in Python can be made. Topology optimization consists of five main steps; the Finite Element Analysis, calculating the sensitivities, applying mesh-independency filters, calculating the constraints and then the optimization, using the Method of Moving Asymptotes. These steps will be explained in chapter 2.3.

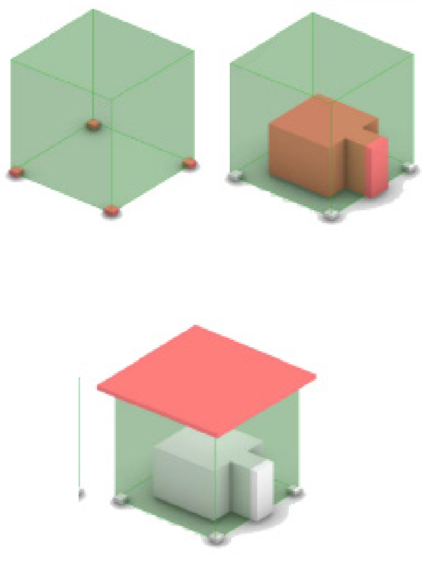


***Figure 7**: Visualization of the inputs by Ivan Avdic*

Then, the self-weight gets implemented in the algorithm. Depending on an material property, the force vector [F] gets another value. This introduces many difficult problems over the procedure by Bendsoe and Sigmund. When the compliance is minimized, the final densities are given as an output.

The last step is to verify these output densities, to see if they show similar results with other software. As this is purely to verify and evaluate the algorithm, the final product will not contain this step. Overall all the computational parts will be written in Python, using libraries like NumPy and SciPy. Verification will be performed by exporting the design space to ANSYS and comparing the results. When this works, the implementation to Grasshopper's IronPython could be made using a proxy server (as Grasshopper does not support NumPy).

>>>1.8 Planning and organization

Figure 8 shows the proposed planning for this research, following the methodology as proposed in figure 5. The first step in this methodology is to create a broad understanding about topology optimization, which was done in the first weeks. After the first objectives were states, research was done to how topology optimization works and how the algorithms work. Before P2, the research framework was written, and with feedback edited in the weeks following P2.

In the weeks after P2 the mathematical methodology was written, to ensure a broad understanding how topology optimization works in every step. Understanding how it works on this level will generate enough knowledge to easily implement this in an algorithm. Also, literature was needed to understand this, but more importantly to implement self-weight.

After week 15 it this methodology is finished and the focus can be placed on the computational methodology. Writing an topology optimization code that contains self-weight is the first objective. When this is working, the translation can be made towards 3D. Abaqus is an program that might make this step easier, but more important, save computational time. Lastly, any architectural lessons of the algorithm will be drawn, and the research questions answered. P4 phase focusses more on implementing the algorithm to toy problems and drawing conclusions therefrom. This building phase is the most important phase as the final product is developed.

After the toy problems are properly designed, conclusions can be drawn from them and the total algorithm is written. When the algorithm, which is the final product, is completed, some test can be run with it in order to answer the research questions.

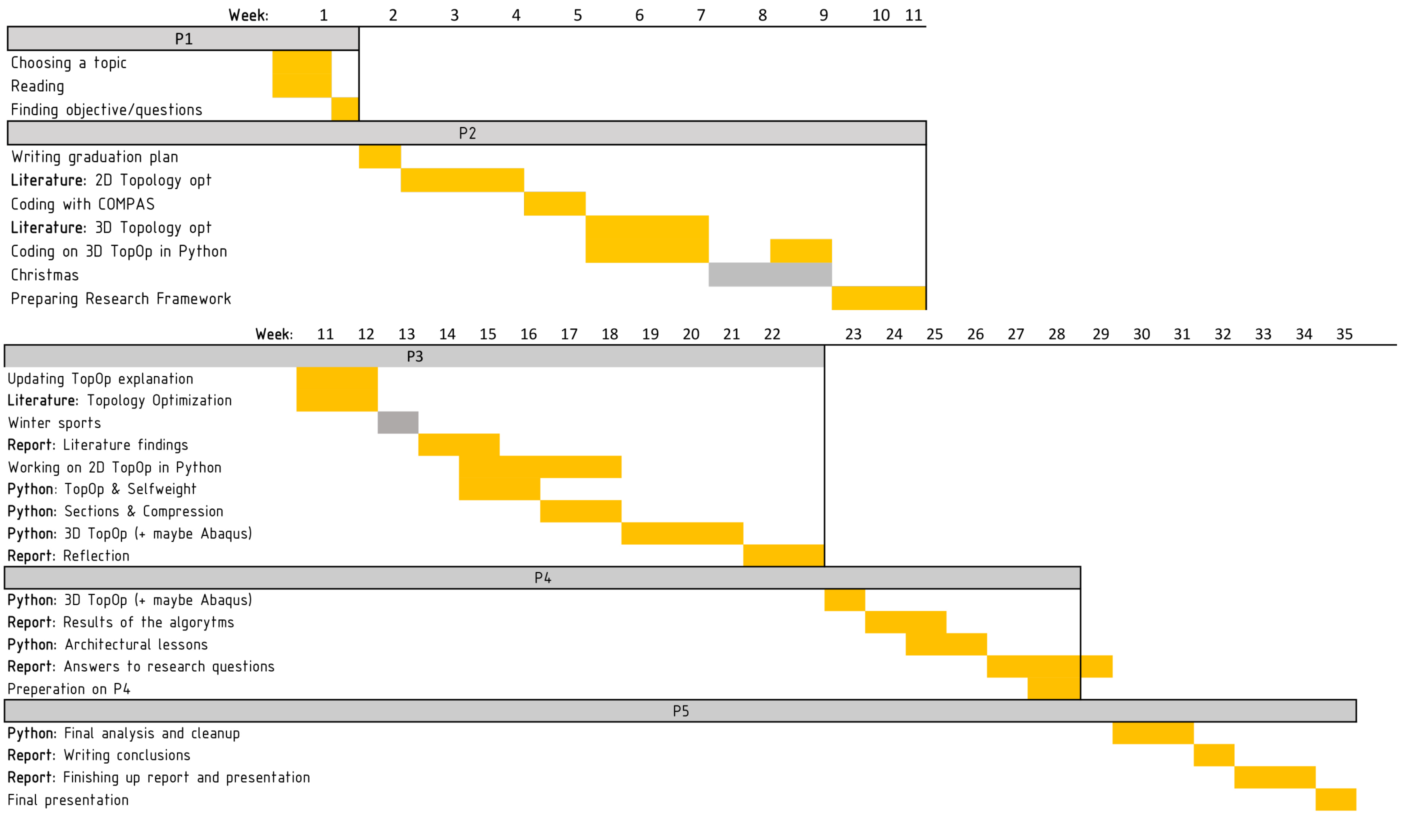


***Figure 8**: Planning of this research*

0 2 T o p o l o g y   o p t i m i z a t i o n

>>>2.1 Introduction

As mentioned in chapter 1, the build phase exist of two different steps, the mathematical design and the algorithm design. This section of the report will follow the mathematical approach on topology optimization. This chapter will establish a foundation on which the later design is based, this foundation will be based on the literature and can also be seen as the literature study. It introduces the most common approach on solving topology optimization, discuss possible applications in architecture and solutions how to make these applications more useful. Creating a structured mathematical basis is key in writing a well working algorithm.

>>>2.2 What is topology optimization?

As mentioned in chapter 1, the build phase exist of two different steps, the mathematical design and the algorithm design. This section of the report will follow the mathematical approach on topology optimization. This chapter will establish a foundation on which the later design is based, this foundation will be based on the literature and can also be seen as the literature study. It introduces the most common approach on solving topology optimization, discuss possible applications in architecture and solutions how to make these applications more useful. Creating a structured mathematical basis is key in writing a well working algorithm.

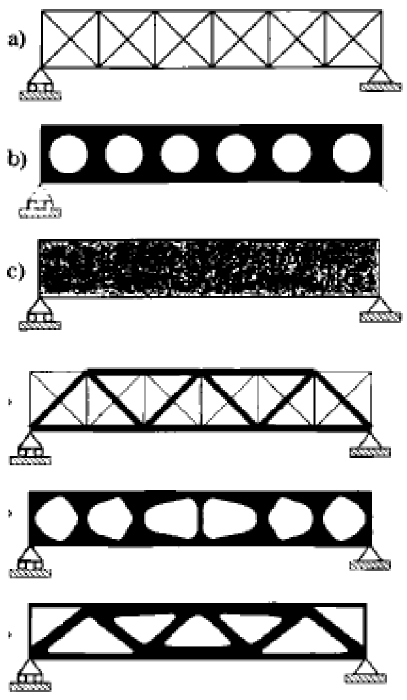


Figure 9: *Structural optimization of a beam*

To understand topology optimization, lets see the different versions op beam optimization. Figure 9 shows a beam that will be structurally optimized, using three different methods (Sigmund, 2001).

Size optimization
Method A (fig 9) increases the size of some elements in this truss structure, following known properties and behavior of these types of beams. The final structure will always follow the prescribed (truss) structure.

Shape optimization
Method B optimizes the shape that is given and is able to generate a new mesh that ensures a maximized stiffness, while still following the defined shape.

Topology optimization
The last method is topology optimization, where the only inputs are the design space and the loads/supports. The method generates the most ideal shape that maximizes the stiffness in this beam. Note that the variable in this beam is the stiffness but many more variants of topology optimization exist. Heat transfer and fluid flow are often-used applications of topology optimization as well, although this research will only focus on structural topology optimization.

>>>2.3 Nomenclature

- **Topology optimization**

Very simplified, topology optimization is the reduction of voxels (or pixels) in a given design space, in order to maximize the stiffness. Figure 10 shows this in a small beam, where the loads and supports are given and an optimized structure is generated. The figure contains the same amount of inputs and outputs as later used problems. Therefore the user inputs are:

- **The design space**

The design space is defined as an area that consists of voxels (or pixels) in which a freeform can be generated. The voxel-size will influence the preciseness of the final result largely.

- **Loads**

The problem should always contain one or multiple loads, or the calculation does not make sense. There is no limit on the amount or direction of the loads. Usually in topology optimization SI-units are not used, as it is a model.

- **Supports**

Lastly the problem contains one or multiple loads. For supports it is important to note that the direction in which the support is attached, has to be configured. A roller support behaves differently than a fixed support.

When these inputs are defined, the first step in topology optimization is completed, namely defining the problem. This chapter will further explain each step more thoroughly. To better understand these explanations, some terms are explained first.

- **Compliance**

For a single spring the compliance is the inverse of stiffness. For the system that is solved in topology optimization it can be described as the degree in which the structure strains due to the applied forces.

- **Penalization power**

A variable that will round values to their closest binary value. The higher the penalization power, the faster this happends. Page 19 will explain penalization a bit clearer.

- **Degrees of Freedom (DoF)**

The amount of directions that a node can move in. In 2D this is 2, horizontally and vertically and in 3D this value is 3 per node.

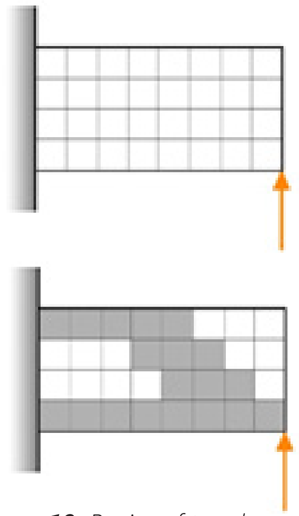


***Figure 10**: Basics of topology optimization*

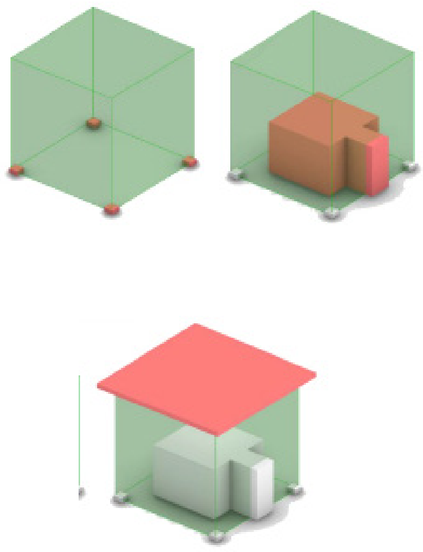


***Figure 11**: Visualization of the user inputs by Ivan Avdic*

>>>2.3.0 Reading guidelines

To assist the reader in understanding this research this chapter will describe how this thesis should be read. As there are a lot of equations and pseudocodes, properly reading them will lead to a better understanding of these.

Equations are written in LaTeX which is the standard notation for mathemetical equations. Each equation has a number to be references with, shown as [x]. In the citations also a paragraph about equations, showing the sources behind all the equations.

Figures are also referenced in one list in the citations sections. Each figure number has its source to be found there.

Flowcharts are used as well, following basic rules of flowcharts. Most notably is that the rectangles corrospond to computational processes, rectangles with rounded edges to human processes and diamonds to inputs. All flowcharts are made using the program Draw.IO.

Lastly pseudocodes have a few rules that need to be known and are specific to this research. When a piece of the algorithm is mentioned in text it will be written in `courierNew`. Further information can be retrieved from how it is written, where red text means an `input`. Bold text represents a **`matrix`** and italic text represent a *`vector`*.

Pseudocode will look like the following:

```
#This is an example of pseudocode

input = 1

for ilist in randomMatrix:
     result = input * ilist
     add result to resultMatrix
```

Listing 1*: Pseudocode name*

The final algorithm can be found in appendix A (2D) and in appendix B (3D).

>>>2.3.1 The procedure of topology optimization

The commonly used methodology of topology optimization is developed by Bendsoe and Sigmund and is often used as the basis for further research. This methodology is shown in figure 12, where the first step is already described in the last paragraph.

The main objective of topology optimization is *"to find the material distribution that minimizes the structures deformation"* (Liu & Tovar, 2014). Usually this is described as minimizing the compliance, which is the mathematical definition of the objective. This can be written as (Sigmund, 2001):

$$\min_{\mathbf{x}} C(\mathbf{x}) = \langle \mathbf{U}, \mathbf{F} \rangle = \mathbf{U}^T \mathbf{K} \mathbf{U} \qquad [0]$$

Or in other worlds, the goal is to minimize the compliance, which is dependent on x. Where x is a vector of all the densities of the elements:

$$\mathbf{x} = [x_i]_{n \times 1}$$

The compliance can be rewritten with U, the global displacement vector and K, the global stiffness matrix. This formula is further described paragraph 2.3.2. This can only be solved when the following are true:

$$subject\ to: \begin{cases} \frac{V(x)}{V_0} = volfrac \\ KU = F \\ 0 < x_{min} \leq x \leq 1 \end{cases} \qquad [1]$$

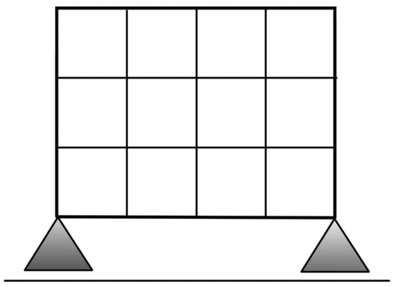


Figure 12: *Example for this explanation*

The first section tells that the volume of the total beam, should always be a set fraction (the user variable *volfrac*) of the design space (V_0). In TO this is the main constraint, which means only results are given that fulfill this formula.

The second section is a very basic formula in physics, namely that the stiffness (*N/mm*) is equal to the force over the displacement. The next chapter will show further how this works on the global scale.

The third section is focusing on x, and how it should be used. Where x is a value per element that somewhat functions like a boolean, it contains a 0 or an 1. A 0 means the voxel is not important for the whole design and can be removed, an 1 means it has great importance. Also, values in between will occur and for these the closer to 1, the more important it is. An important note is that x can never be 0, to avoid singularity and unsolvable fractions, but 0 will be replaced with a very small x_{min}, usually around 0,001.

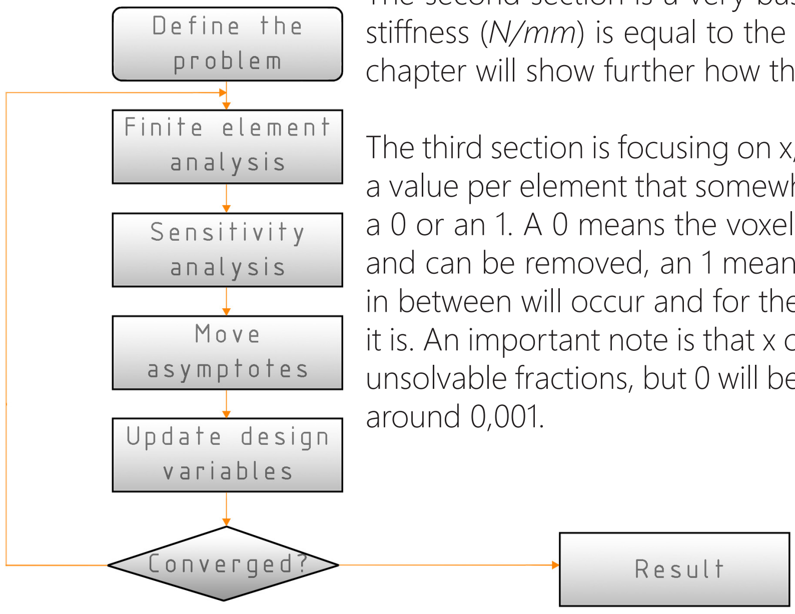
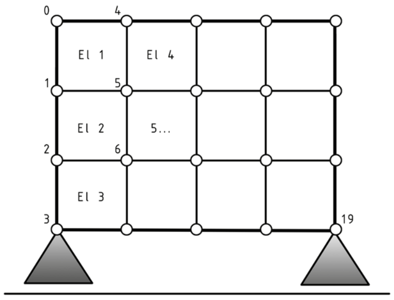


Figure 13: *Procedure of topology optimization (Sigmund, 2001)*

>>>2.3.2 Finite element analysis

The first step in topology optimization is the finite element analysis, where the displacements of each element are calculated. This paragraph will cover what happens in this step. To calculate what happens in each element when under pressure is the very basis of topology optimization. Finite element analysis is the simulation of a physical phenomenon using numerical computing on a discrete design space, where this research focusses on a load and its stresses. To explain how finite element analysis works a small example is presented in figure 14 that is based on a pixeled beam.



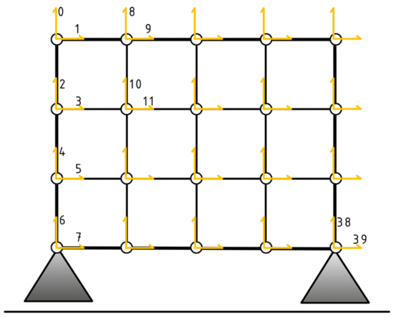


Figure 14: *Numbering of the example*

Numbering

The first important thing to note is how each element and node are numbered. Each element (pixel or a voxel) will be numbered from top to bottom and left to right. These numbers aren't specifically important, but for computational reasons and for explanations, they will receive an index. What is more important is the nodes of the elements. Each element will be defined as a space between 4 (in 2D) or 8 (in 3D) nodes, each with it's own index. The numbering of these nodes is important, as it influences the efficiency of the algorithm. This will be of later focus and can be read in chapter 3.4. Each node will receive an index, but also has degrees of freedom. These degrees of freedom will be used later on in the calculations in stead of the nodes themselves. It is important that the numbering of the DoFs follows the index of the nodes. Another way of numbering is to write the index and the direction, such as $U_{1,x}$ and $U_{1,y}$. This is harder to program in a later phase, so the DoF index is used.

Finite element analysis

The main idea behind FEM is to calculate the displacement in each node, using the stiffness of this node and the force(s) that work on the space.

$$KU = F \qquad [2]$$

Or: The stiffness times the displacement is the force. This also works for matrices, when they are configured correctly. The following paragraph will go over the construction of the individual matrices, their shape and their values.

The displacement vector U

For each degree of freedom the displacement should be calculated, to give a good overview how the node behaves. Therefore, a vector should be created with the size of the degrees of freedom; or:

$$[1 \times \ ((nelx{+}1)\cdot(nely{+}1)\cdot(nelz{+}1))] \qquad [3]$$

The vector is filled with unknowns, namely the displacement at each DoF. For some DoFs the displacement is known to be 0, at the supports. A cylinder support only has this at the vertical DoF, a fixed support for both DoFs. For the example that was created in figure 14, the displacement vector has 0 in DoFs 6,7,38 and 39.

The force vector F

The force vector is defined as the prescribed force at each DoF. In most cases, the value of the force is 0 at the DoF, except for the DoF where a force is directly applied. It has an equal size as [U], as it is also a vector and the length is the size of the DoFs. Note that not only one force can be applied, but theoretically each DoF can have a force. Distributed loads should be divided over the DoFs, also shown in figure XX.

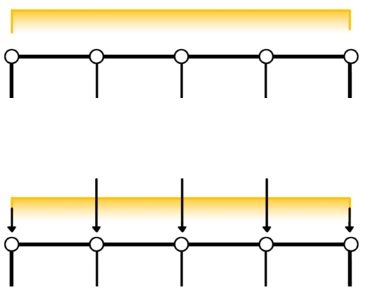


Figure 15: *Distributed load F*

The K-matrix

The last piece of the FEM is the most challenging to solve, as the x, the density of an element, plays a role in this matrix. To explain how this works, first let us take a look at one element. Each element has its own element stiffness matrix, K_e, which defines the behavior of the element. The construction of K_e is based on shape functions, which are described by de Orio as functions that "... interpolates the solution between the discrete values obtained at mesh nodes" (de Orio, 2008). Figure XX shows element nodes and their shape functions. At a node, the function is always 1, while being 0 at other nodes. From these shape functions the element stiffness matrix can be retrieved, however the derivation of the matrix or how these shape functions determine the K_e falls outside the scope of this research.

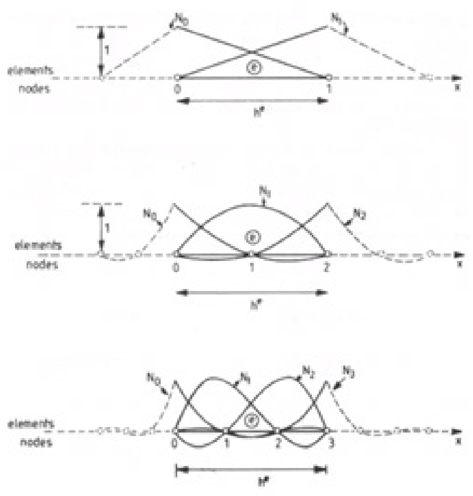


Figure 16: *Examples of shape functions depending on the nodes*

The K_e would look something like figure 18, containing values in an 8x8 matrix (note that for 3D the element stiffness matrix is of size 24 x 24, as each node has 3 DoFs). When deriving element stiffness matrices, the values are stiffnesses, often in the form of EA/L. In topology optimization the size of each element is assumed to be 1 and also the Young's modulus does not need to be definite. For this research, K_e matrices will be used that are created by Sigmund (2001) and Liu and Tovar (2014), they can be found in appendix A. These matrices are not dependent on Young's moduli, but on the poissons ration, which is equal for all elements.

The K_e will be constructed for each element, and multiplied by its density. This is important, as an element with density 0, should not influence the stiffness of the whole structure. When multiplying with the density, the penalization power, p, is also taken into account. This is an input variable, usually between 1 and 3, that influences how much non-binary values are allowed. The higher the penalization power, the more the algorithm gets punished for using non-binary values.

The formula to calculate the K_e is now:

$$K_e = x^p \cdot K_e \qquad [4]$$

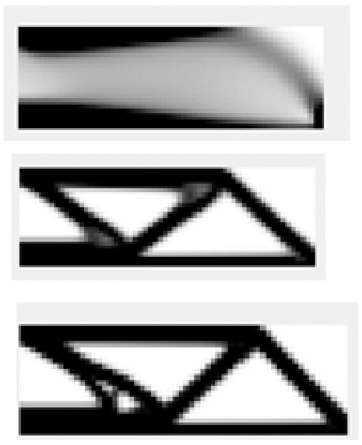


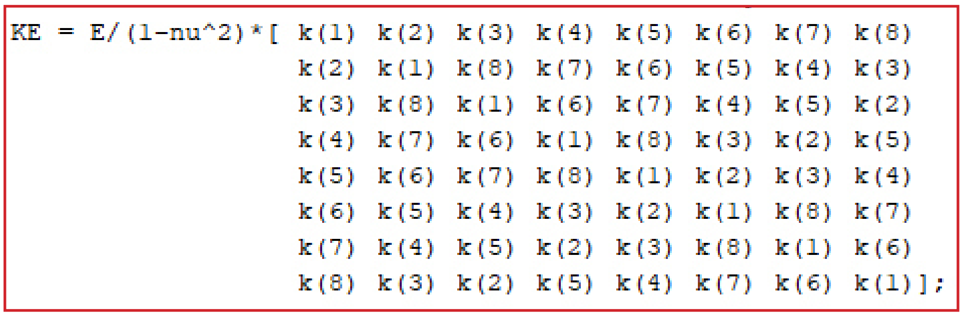
Figure 17: *Influence of a higher penalization power*

k(1)	= 1/2 - v/6
k(2)	= 1/8 + v/8
k(3)	= -1/4 - v/12
k(4)	= -1/8 + 3v/8
k(5)	= -1/4 + v/12
k(6)	= -1/8 - v/8
k(7)	= v/6
k(8)	= 1/8 - 3v/8

with *v* as the poisson ratio

Figure 18: *Values for k*

In the example the focus will be on element *i* and its K_i. Following the matrix by Sigmund, found at appendix A, the matrix will look like the following:

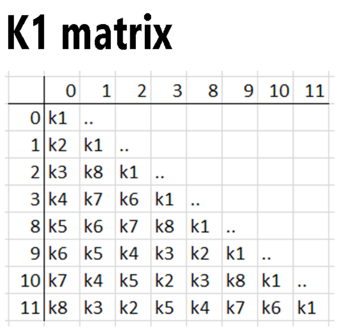


```
KE = E/(1-nu^2)*[ k(1) k(2) k(3) k(4) k(5) k(6) k(7) k(8)
                  k(2) k(1) k(8) k(7) k(6) k(5) k(4) k(3)
                  k(3) k(8) k(1) k(6) k(7) k(4) k(5) k(2)
                  k(4) k(7) k(6) k(1) k(8) k(3) k(2) k(5)
                  k(5) k(6) k(7) k(8) k(1) k(2) k(3) k(4)
                  k(6) k(5) k(4) k(3) k(2) k(1) k(8) k(7)
                  k(7) k(4) k(5) k(2) k(3) k(8) k(1) k(6)
                  k(8) k(3) k(2) k(5) k(4) k(7) k(6) k(1)];
```

It can be seen that this matrix is symmetric and only has 8 different values, a 3D matrix has a different approach on building this, which can be seen in appendix B, lines 71 - 119. This makes hardcoding easy and usually this is performed in topology optimization.

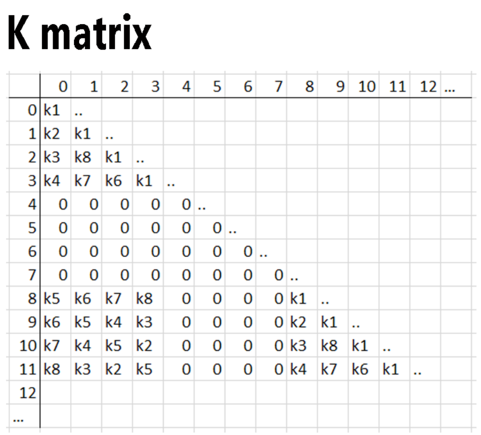
Now that the element stiffness matrix, K_e, is created, the next step is to create the global stiffness matrix K. Each element has its own K_e and they have to be properly placed into the larger global stiffness matrix. This matrix is a squared matrix with equal length as the DoFs. To place the K_e in K, a coordinate system is used, with the DoFs as coordinates. For example, in K_1 in figure 14, the DoFs are known to be [0, 1, 2, 3, 8, 9, 10, 11]. $K_{e,1,1}$ = 0.495 and $K_{e,9,10}$ = 0.0137. These coordinates pass on in the global stiffness matrix, as seen in figure 19. A lot of DoFs exist in more than one element, in that case the values can be added. This process results in the global stiffness matrix K.

K1 matrix



	0	1	2	3	8	9	10	11
0	k1	..						
1	k2	k1	..					
2	k3	k8	k1	..				
3	k4	k7	k6	k1	..			
8	k5	k6	k7	k8	k1	..		
9	k6	k5	k4	k3	k2	k1	..	
10	k7	k4	k5	k2	k3	k8	k1	..
11	k8	k3	k2	k5	k4	k7	k6	k1

K matrix



	0	1	2	3	4	5	6	7	8	9	10	11	12	...
0	k1	..												
1	k2	k1	..											
2	k3	k8	k1	..										
3	k4	k7	k6	k1	..									
4	0	0	0	0	0	..								
5	0	0	0	0	0	0	..							
6	0	0	0	0	0	0	0	..						
7	0	0	0	0	0	0	0	0	..					
8	k5	k6	k7	k8	0	0	0	0	k1	..				
9	k6	k5	k4	k3	0	0	0	0	k2	k1	..			
10	k7	k4	k5	k2	0	0	0	0	k3	k8	k1	..		
11	k8	k3	k2	k5	0	0	0	0	k4	k7	k6	k1	..	
12														
...														

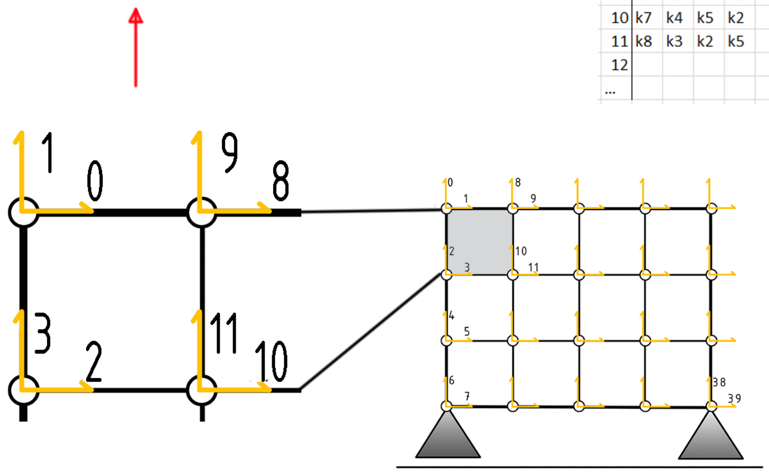


Figure 19: *Translation from* K_e *to the global stiffness matrix K*

Solving the system
When the three elements are properly configured, the solution can be found by solving *F = KU*. The global stiffness matrix K is at this moment fully symmetric and singular, therefore no solution can be found. It is important that some values of [U] are 0 (because of supports) so a solution can be found. By setting these values to 0, values will be removed from K, allowing for the solve. The solve is performed by solving Ax = b, which is a common formula to solve. Doing this by hand is a very hard and tedious process, so this is performed by the computer using preset functions. Later chapters will cover these solvers. The output will be a displacement vector with the displacements for each DoF.

The Compliance
Now that we have the displacement vector U and the stiffness matrix K, the compliance can be calculated, as it was formulated in equation [0]. It makes sense that the equation to calculate the compliance is [0], as it is the inverse of K. K is the same as $(FU)^{-1}$, or, when inverting the matrices, U^TF.

Another notation of the total compliance, is given in equation [5], which is noted as the som of the compliance at each DoF. In this new equation, also [4] is combined to create the compliance with less equations. Note that in this equation, the K is not the global stiffness matrix, but the element matrix, K_e.

$$C = \sum_{e=1}^{nele} -x^p U_e^T \; K_e \, U_e \qquad [5]$$

As this is the “first” iteration, this will give the compliance for the system, where all the densities are equal and there has been no optimization. In order to optimize the system, the densities need to be changed and the compliance minimized. The next paragraph will further elaborate on the method of finding the values for x where the compliance is minimal.

>>>2.3.3 Sensitivity analysis

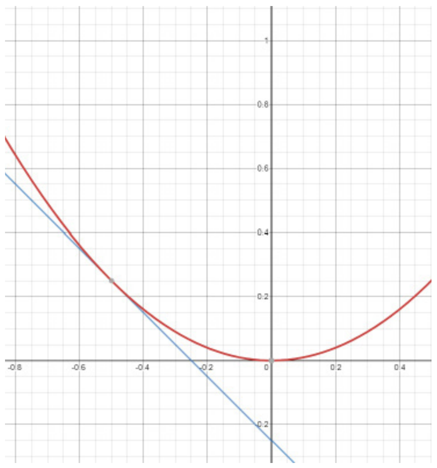


Figure 20: *f(x) with the derivative at 0.5*

In this next step the sensitivity analysis is performed. This means that for each element it needs to be known how much it will influence the compliance. Let's pretend the compliance functions like a random function f(x) and looks like figure 20. When looked at a value of x, the slope of f(x) will tell how much the result will change when x changes. This also applies for the compliance, when the slope of C(x) is large, the final compliance will change a lot when the value of x is changed. Or: when the derivative of C(x) is large for a value of x, this element is important to the structure and therefore has a high chance of being placed.

The derivative of the compliance, [6] is given by (Bendsøe & Sigmund, 2004) as follows:

$$\frac{\partial C}{\partial x} = -p \cdot x^{p-1} U_e^T \ K_e \ U_e \qquad [6]$$

This results in a matrix with size of nele, but in stead of the compliances, it gives the influence on the compliance for each element.

Mesh-independency

It is important to ensure that the results are independent of the mesh and filtering the sensitivities is a efficient way of solving this. "*This means modifying the design sensitivity of a specific element, based on the weighted average of the element sensitivities in a fixed neighborhood*" (Bendsøe & Sigmund, 2004). This means that the sensitivities will be updated, after they are calculated but before optimizing, which will go as follows:

$$\frac{\partial C}{\partial x_k} = \frac{1}{x_i \sum_{i=1}^{nele} H_i} \cdot \sum_{x^i}^{nele} H_i x_i \frac{\partial C}{\partial x_i} \qquad [7]$$

In this formula, H_i is the weight factor and nele the number of elements. Note that this is the the calculation for element k, and in the formula the density of its neighbour is calculated. Therefore element x_i is dependent on its neighbor, x_k. H_i is defined as follows:

$$H_i = r_{min} - dist(k, i) \qquad [8]$$

R_{min} is an input variable that is the radius in of this weighted average. The distance is the straight line distance between two elements. When the elements are further away from each other than R_{min}, nothing is added, as they are not neighbors.

>>>2.3.4 Optimality criteria

To find the new values of x_{new}, a classical approach is taken called optimality criteria (OC). This method assumes a certain optimal condition, called B_e, that is optimal when $B_e = 1$, where:

$$B_e = -\frac{\partial C(x)}{\partial x_e}(\lambda \frac{\partial v(x)}{\partial x_e})^{-1} \quad [8]$$

In here, λ is the Lagrange multiplier, that is associated with constraint *v(x)*. Bensoe proposed the following method to update the value of $x_{new,}$ depending on $X_e B_e$:

$$x_e^{new} = \begin{cases} if: \quad x_e B_e^\eta \leq max(0, x_e - m): & max(0, x_e - m) \\ if: \quad x_e B_e^\eta \geq min(1, x_e - m): & min(1, x_e + m) \\ \qquad\qquad else: & x_e B_e^\eta \end{cases} \quad [9]$$

In this formula, m is a certain (positive) move-limit and η is a numerical damping coefficient. Usually in minimum compliance problems, values of m = 0.2 and η = 0.5 are used. Because the Lagrange multiplier is the only unknown, the constraint can be ignored and B_e can be rewritten as:

$$B_e = -\frac{\partial C(x)}{\partial x_e}(\lambda)^{-1} \quad [10]$$

To find the value of the Lagrange multiplier, the bisection method is used to find the value of λ. To do this, two values of λ are chosen, both on the outside of the possible values of λ. For minimum compliance problems, $\lambda_1 = 0$ and $\lambda_2 = 10^5$ are enough. In the bisection method, λ_{mid} is initialized, being the average of λ_1 and λ_2. The values of x_e are updated with this new Lagrange multiplier, resulting in new values of x_{new}. One of the constraints in chapter 2.3.1 was that:

$$\frac{V(x)}{V_0} = volfrac \quad [11]$$

so therefore, the Lagrange multiplier is correct when the sum of x_{new} is equal to *volfrac * V(0)*. A loop is started that halves the domain to approach the correct value of λ. When the difference between λ_1 and λ_2 is small, usually something like 0.001, the optimal values of x_{new} are found.

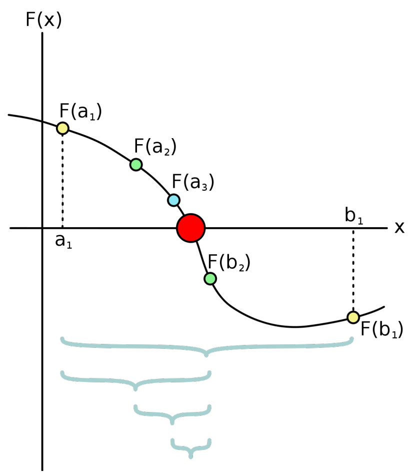


Figure 20: *Bisection method*

>>>2.3.5 Updating the design variables

After the values of x_{new} are found, these can be implemented again at the start of the loop. A new FE analysis is performed with the new values of **x**. Slowly the compliance should drop to a lower and lower level. When the values of x are not changing anymore (or after a set amount of loops) the algorithm stops and gives the final values of x. These are the densities of each element and therefore the likeliness that have to exist. Some postprocessing can be done to show the grayscales better, but this is seen as unnecessary for this chapter.

>>>2.4 Topology optimization in Architecture

The previous chapter explained how the standard procedure, following Bensoe and Sigmund, is performed. This research will focus on the question *"How can we design structures for masonry buildings using topology optimization?"*, so the translation to architecture has to be made. How could architecture take advantage of the possibilities of topology optimization? This chapter will shortly cover any approaches found in literature that could be implemented in topology optimization for architectural models.

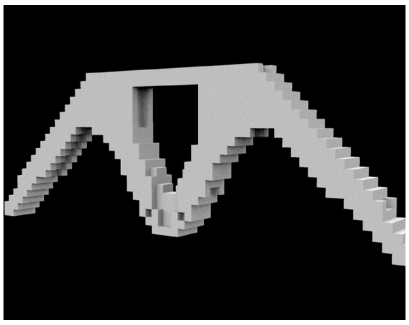


Figure 21: *3D topology optimization*

3D topology optimization

Chapter 2.3 spoke about topology optimization in the 2D field, with a design divided into pixels. As architecture is a research field that focusses on 3D challenges, this translation is an important step towards the main objective of this research. Translating from 2D to 3D does not change in the methodology, the math behind 3D topology optimization is the same behind 2D. However, some changes have to be made to create a working 3D model.

Selfweight

The implementation of self-weight makes a lot of sense to introduce into topology optimization. Usually, the self-weight of each element is very small in comparison with the force applied on the design. In architecture however, each element is the main source of forces and therefore, placing an element always influences the forces distributed in the design. This asks for a more complicated algorithm, as another variable is introduced in the system. This research will focus mainly on masonry buildings, where self-weight is even more important. It is expected that implementing self-weight in the methodology, will result in compression-only methods. This translates the system easier into a masonry building. A lot can be found in literature about the selfweight, so this will be described in the following chapter.

Snowload

Buildings will always have a force on its roof, caused by diverse forces, but usually this is mainly caused by snow. This force is not set on a specific place, but will be placed on the roof. This force is dependent on the total shape and requires a similar approach as the selfweight. Implementing snowload isn't seen as part of the literature study and will therefore be discussed in chapter 3.5.

Roof constraint

Usually there are very limited direct forces that are present in the design space. Topology optimization can only function properly if there are at least forces to optimize the system with. One constraint that can be added is the fact that each void has to have a closed roof above it. Adding this constraint will allow the algorithm to create geometry, without preset forces. As this is also seen as a product of this research, it will be discussed in chapter 3.5.

>>>2.5 Density dependent forces in topology optimization

Implementing density dependent forces follows the same procedure as previously described. A design space consisting of elements is created with supports and loads. Both the K-matrix and the displacement vector U are created on the same ways as in chapter 2.3.2. The main difference is that the force vector F gets created each iteration of its own. For each DoF, the force of the selfweight has to be calculated by multiplying the density of an element by a certain force value. This force should depend on the size of the grid in comparison to the force, something that will be figured out later. Figure 22 shows an element, x_i, that has a density of 1. The elements nodes will have ¼ of the forces in the y direction. The forces at the DoFs are added and the force vector is created.

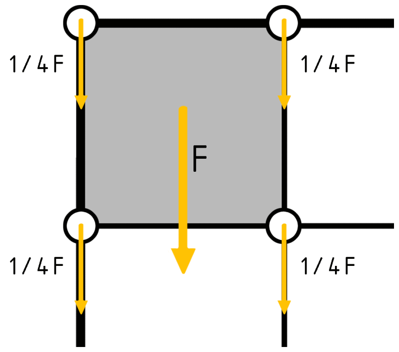


Figure 22: *Element x_i with self-weight*

Sensitivity analysis

Bruyneel and Duysinx (2001) described how to perform the sensitivity analysis when applying density dependent forces. As there are multiple variables, the sensitivity analysis from chapter 2.3.3 does not apply anymore. Instead, the following steps have to be taken to get the sensitivity of the compliance.

Given is that the compliance is:

$$C = U^T K U \quad [12]$$

Also, it's known that *KU* = *F*, so the compliance can be rewritten as:

$$C(x_e) = F^T U + \lambda(KU - F) \quad [13]$$

where λ is a lagrange multiplier that will be defined later, but where *λ(KU – F)* = 0.

When taking the derivative of this function:

$$\frac{\partial \widetilde{C}}{\partial x} = \frac{\partial F^T}{\partial x} \cdot U + F^T \cdot \frac{\partial U}{\partial x} + \lambda^T \left(\frac{\partial K}{\partial x} \cdot U + K \cdot \frac{\partial U}{\partial x} - \frac{\partial F}{\partial x}\right) \quad [14]$$

Using the chain rule to be able to choose λ, so that δU is removed from the equation:

$$\frac{\partial \widetilde{C}}{\partial x} = (F^T + \lambda^T K)\frac{\partial U}{\partial x} + \frac{\partial F^T}{\partial x} \cdot U + \lambda^T \left(\frac{\partial K}{\partial x} \cdot U\right) - \lambda^T \frac{\partial F}{\partial x} \quad [15]$$

if $F^T + \lambda^T K = 0$, then δU is removed. It is known that *KU* = *F*, so λ is chosen as λ = - *U*. Then the derivative is:

$$\frac{\partial \widetilde{C}}{\partial x} = \frac{\partial F}{\partial x} \cdot U - U^T \frac{\partial K}{\partial x} U + U^T \frac{\partial F}{\partial x} \quad [16]$$

The derivative from equation [x] can be rewritten as:

$$\frac{\partial \widetilde{C}}{\partial x} = 2U^T \frac{\partial F}{\partial x} - U^T \frac{\partial K}{\partial x} U \quad [17]$$

When looked at this sensitivity, it can be shown that when F is not dependent on x, the sensitivity is U^T δK U, which is conform the previous chapter, see equation [5]. Now that the force is density dependent, the sensitivity can be rewritten as:

$$C(x_e) = -px^{p-1}U_e^T K_e U_e + U_e^T F_e \qquad [18]$$

This is where the main problem of self-weight comes in. Equation [x] is always negative and behaves monotonic and can therefore be easily analyzed. In the topology optimization code of Bensoe and Sigmund, this property is exploited to generate easy and fast solutions. This also makes sense, as equation [18] is technically an equation of *f(x) = x^p*, which will always have an easy to find minimum, as long as p is inside the domain of [1,3]. In the case of equation [19] the force is dependent on the density and this creates harder to generate solutions. Equation [19] can be both positive and negative, so a change in the density of an element, can result in a larger compliance. This asks for more complicated ways of finding the minimum compliance (Bruyneel & Duysinx, 2001).

Finding the minimum compliance for non-monotonic functions
The non-monotonic behaviour of the function asks for a different solver than when there are no density dependent forces. This solver is found in a very popular optimizer in topology optimization, called the Method of Moving Asymptotes (MMA). MMA is “a method of non linear programming in (structural) optimization, characterized by an iterative process where a new strictly convex subproblem is generated and solved per iteration” (Blackman & Miller, 2014). MMA follows the following steps in order to solve a function f(x):

The first step is an initial guess, on iteration k = 0, where the value of f(x_k) is returned. Then the gradients for the constraints are calculated. Next, an subproblem is generated, that exists of convex functions, based on the gradients for the constraints and on information of the previous iteration. This subproblem can now be solved, which results in a new point. This point is now the new, local, optimum and will be x_{k+1} (Svanberg, 1987).

Figure 23 shows an objective function (black) and the initial guess x_0. From the x_0 a subproblem is generated in the blue line. The optimizer adjusts the asymptotes in order to create the red line, which is the convex approximation. This subproblem is solved in order to find the new local optimum. For the second iteration the previous solution is taken, until the convergence criteria is met (Blackman & Miller, 2014).

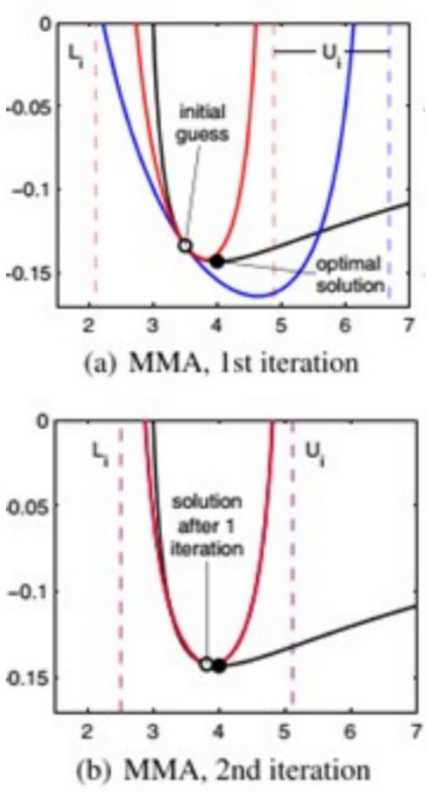


Figure 23: *Simplification of the MMA*

03 Algorithm design

>>>3.1 Introduction

The building phase of this process exists of two methodologies, which are the mathematical design and the algorithm design. The main difference is that the mathematical design focusses on the techniques and math behind Topology Optimization and its theoretical methodology. This chapter will build further on that foundation, translating it into useful code. There are already some known algorithms that form the basis of further adjustments towards an algorithm that is more broadly applicable on architecture. There is a standard structure, developed by Sigmund, that is usually taken as the basis for Topology Optimization code. This structure and its lines will be discussed and explained. Then, other algorithms that are used in this process are described and looked at. These algorithms mainly contain methods of optimization and implementing 3D in Topology Optimization. Then, the methodology is described to include architectural models in topology optimization. For this process, 6 fictional case studies are created, each with it's own problems to solve. Through these cases, most of the architectural challenges should be solved.

>>>3.2.1 A 99 line topology optimization code

One of the first papers that was published about a Topology Optimization code, was written by Sigmund. His paper includes a 99 line code that solves topology optimization problems in the 2D field. The paper creates a standard procedure that has already been discussed at chapter 2.3.1. The procedure that Sigmund proposed, has been the basis for many other variants of the script. To understand later changes and variants on the script, lets first look at the structure of this script. Sigmund wrote the script in MATLAB, but this thesis will focus on Python as programming language, due to the implementation in Grasshopper.

Several translations have been made to Python. Initially the version of COMPAS was used, written by the ETH Zurich. COMPAS is a python library that is created for Rhino and Blender, to implement topology optimization, dynamic relaxation and several other numerical process. COMPAS' implementation with Grasshopper geometry is a bit harder, as most geometry has to be translated into specific COMPAS geometry (COMPAS, 2020). As the goal of this research is to write a Grasshopper-specific, this can be greatly improved. The input of the algorithms are matrices, and it is more efficient to write the translation in Python.

Another Python implementation is written by Liu and Tovar, where they add 3D into their own translation of Sigmund's procedure. This code is focussing on the 3D translation, but also includes some improved lines for faster calculations (Liu & Tovar, 2014).

The last Python implementation forms the basis of the code of this script. It is written by Arjen Deetman in combination with a MMA solver. Chapter 2.6 already explained the need of another solver, in stead of Optimality Criteria. Deetman developed the MMA solver for Python and also added a topology optimization script. This script is a translation of Sigmund's script with the addition of the MMA solver. This script will be described in pseudocode in the following chapter.

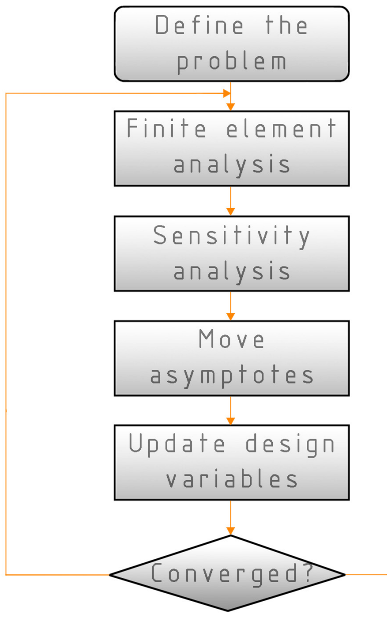


Figure 24: *Stages of the algorithm*

>>>3.2.2 Explanation of the code

This chapter is included in this thesis, as it is meant to open the black box and explain what happens in the code. The code that Arjen Deetma developed exists of 195 lines, including both the Optimality Criteria and MMA solver. The main program of the solver follows several stages to solve the topology optimization problem. These stages are shown in the figure 24.

User input

The first stage is always the user input and translating it into useful data. Eventually all of the input should be translated into the matrices. There are several possible inputs to solve the problem. The first input is always the amount of elements in the x and y direction (`nele`, `nely` *and* `nelz`). The second input are the `volume fraction`, the `minimum radius` (see eq. 8) and the `penalization power`. When these values are known, matrices can be created of the right size to use later on, see Pseudocode 1.

The values `nele` and `nDof` are created and are respectively the number of elements and the number of degrees of freedom. It has to be noted that chapter 2.3.2 explains the rules of numbering, which is why in 3D, this value has 3 DoF's at each node. An vector with the `volfrac` is created for *x*, as the final sum of the elements should be equal to a fraction of the volume. The empty array **`dc`** is created and will be filled later with the sensitivities.
The second piece of user input is usually embedded in the code. To solve the FEA, forces and supports are needed. The forces and supports working in the design are vectors with the length of `nDof`. For forces, the force is indicated with its strength at the degree of freedom. This can work in both vertical and horizontal directions and usually is determined with a 1. A list of supports is created as a vector *`fixed`*, and also results in a list of *`free`*, the DoF's that are free to move.

```
x = volfrac.repeat(nele)
dc = makeMatrixOfZeros(shape =(nelx,nely))
#Create supports
fixed = [support1, support2, .... supportn]
free = listOfDofs.remove(fixed)
#Set forces:
f = emptyVector(shape = nele)
f[force1, force2, ... forcen] = 1
```

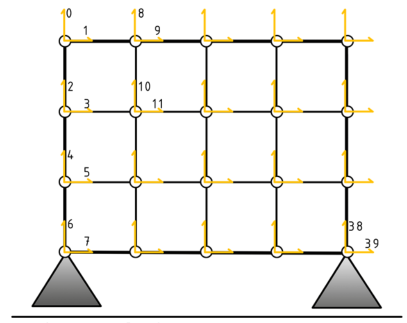
Pseudocode 1: *Configuration of user input*

Initializing

The next step is to initialize the user input and develop some matrices and vectors that are needed to start the loop. As figure 24 showed, the following steps are the FEA and the sensitivity analysis. For the FEA the stiffness matrix needs to be created. Chapter 2.3.2 showed that this is usually done with shape functions, but this is very hard to calculate and asks for more computational time. Usually the element stifness matrix is hard-coded into the algorithm, this algorithm calls the *lk()* function. This generates a 8 x 8 element stiffness matrix (or 64 x 64 in a 3D problem). Both the hardcoded KE matrixes can be found in appendix A.

```python
def lk():
    E = 1
    nu = 0.3
    k = np.array([1/2-nu/6,1/8+nu/8,-1/4-nu/12,-1/8+3*nu/8,-1/4+nu
    /12,-1/8-nu/8,nu/6,1/8-3*nu/8])
    KE = E/(1-nu**2)*np.array([ [k[0], k[1], k[2], k[3], k[4], k
    [5], k[6], k[7]],
    [k[1], k[0], k[7], k[6], k[5], k[4], k[3], k[2]],
    [k[2], k[7], k[0], k[5], k[6], k[3], k[4], k[1]],
    [k[3], k[6], k[5], k[0], k[7], k[2], k[1], k[4]],
    [k[4], k[5], k[6], k[7], k[0], k[1], k[2], k[3]],
    [k[5], k[4], k[3], k[2], k[1], k[0], k[7], k[6]],
    [k[6], k[3], k[4], k[1], k[2], k[7], k[0], k[5]],
    [k[7], k[2], k[1], k[4], k[3], k[6], k[5], k[0]] ]);
    return (KE)
```

Pythoncode 2: *Defenition to create the element stiffness matrix*



```
print edofMat

[2  3  10  11  8  9  0  1]
[4  5  12  13  10  11  2  3]
[6  7  14  15  12  13  4  5]
[10  11  16  17  18  19  8  9]
...
[30  31  36  37  38  39  29  28]
```

Figure 25: *Example of edofMat for the beam*

To create the global stiffness matrix, a matrix needs to be created with the DoF indexes, corresponding to the element index. Following the node numbering of chapter 2.3.2, a matrix is generated with all the DoF's per element index. This is retrieved by taking the 4 nodes of the element en 4 nodes of the element on the right of it. The order of the values is very important for the numbering. For the example the edofMat of this beam is printed in figure 25.

```
edofMat = makeMatrixOfZeros(shape=(nele,8))
for each element:
    ID = (nely + 1) * xvalue * yvalue
    $ID_{right}$ = (nely + 1) * (xvalue + 1) * yvalue
    edofMat =[2*ID+2, 2*ID+3, 2*$ID_{right}$+2, 2*$ID_{right}$+3,
             2*$ID_{right}$, 2*$ID_{right}$+1, 2*ID, 2*ID+1]
```

Pseudocode 3: *Creating the edofMat array*

Another matrix that is initialized before the loop starts, is the H matrix. The H matrix is important to ensure mesh-independency, also as stated in chapter 2.3.3. This chapter showed the weight factor is equal to:

$$H_i = r_{min} - dist(k, i)$$

The H matrix is generated using a coo matrix, with is a matrix based on coordinates. Using a complicated for loop, the proper row and column coordinates are created. The value is stored in the `sH` vector, which is based on the `rmin`. Chapter 2.3.3 showed that the weight factor is equal to the rmin minus the distance between two elements. In the lines, this is shown in line 7.

```python
1  # Filter: Build (and assemble) the index+data vectors for the
   coo matrix format
2  nfilter = int(nelx*nely*((2*(np.ceil(rmin)-1)+1)**2))
3  iH = np.zeros(nfilter)
4  jH = np.zeros(nfilter)
5  sH = np.zeros(nfilter)
6  cc = 0
7  for i in range(nelx):
8      for j in range(nely):
9          row = i*nely+j
10         kk1 = int(np.maximum(i-(np.ceil(rmin)-1),0))
11         kk2 = int(np.minimum(i+np.ceil(rmin),nelx))
12         ll1 = int(np.maximum(j-(np.ceil(rmin)-1),0))
13         ll2 = int(np.minimum(j+np.ceil(rmin),nely))
14         for k in range(kk1,kk2):
15             for l in range(ll1,ll2):
16                 col = k*nely+l
17                 fac = rmin-np.sqrt(((i-k)*(i-k)+(j-l)*(j-l)))
18                 iH[cc] = row
19                 jH[cc] = col
20                 sH[cc] = np.maximum(0.0,fac)
21                 cc = cc+1
22 # Finalize assembly and convert to csc format
23 H = coo_matrix((sH,(iH,jH)),shape=(nelx*nely,nelx*nely)).tocsc
   ()
24 Hs = H.sum(1)
```

Pythoncode 4: *Creating the mesh-independency filter (Deetman, 2019)*

Lastly empty matrices for the displacement U, the total compliance `ce` and the sensitivities dc are created with the element sizes. Also, the loop counter and the elasticity values are set.

```
#Set loop counter and gradient vectors
loop = 0
u  = zeros(size = ndof)
dv = ones(size = nele)
dc = ones(size = nele)
ce = ones(size = nele)
```

Pseudocode 5: *Setting up values before the iteration*

Finite Element Analysis

Now the loop starts, each loop first the Finite Element Analysis is performed, to calculate the displacements. To perform this, the global stiffness matrix has to be created, which is done by first calculating the element stiffness matrix for all the elements, sK. Then, the stifness matrix, $\mathbf{K}$, is assembled into a matrix, where iK and jK are the row and column indexes. Lastly, the constrained DoF's are removed from K, to only take values that are used. Now that $\mathbf{K}$ and F are known, the displacements of each DoF can be calculated into U, where the fixed DoF's are replaced with 0's.

```
#Build the row and column indeces
iK = repeatEntireMatrix(edofMat, 8)
jK = repeatEachValue(edofMat, 8)

#Build the K-matrix, see 2.3.2
K = makeCooMatrix(rows=iK, columns=jK, data=KE*x^p)
K = K.removeOtherThan(free)

U[free] = solve(K, f)
```

Pseudocode 6: *Solving the system*

Sensitivity analysis
Now that all the matrices are known and calculated in the loop, the objective and the sensitivities can be calculated. The objective is to find the lowest total compliance, that is achieved with the set volume fraction. To find this, the gradient of the compliance has to be calculated for each element. The sensitivity and the compliance are calculated and the main input for the solver.

```
#Objective and sensitivity
ce = (U[edofMat].transform * KE) * U[edofMat]
ce = sumof(ce, axis=horizontal)
obj = sumof( $x^p$ * ce)
dc = - p * $x^{p-1}$ * ce
```

Pseudocode 7: *Calculating the obj (compliance) and dc (sensitivities)*

Solving the system
Deetman's code includes 2 different solvers, Optimality Criteria and the Method of Moving Asymptotes. The method of OC has been briefly discussed in chapter 2.3.4, and is usually a faster way of solving the system. MMA is a slower solver, but is not limited to monotonous functions. Both need their own setup, but will return the values chosen for x, which are the input for the next loop.

>>>3.2.3 3D Topology optimization

Chapter 3.2.1 already spoke of the 3D topology optimization code, written by Liu and Tovar. This code, written in MATLAB, forms the base of the translation to the 3D field. Liu and Tovar show that this translation is possible to make, but requires an good indexing system. As the system transitions from 2D to 3D, also many matrices will translate into 3D. When working with Python, this has to be handled properly. Liu and Tovar have rewritten Sigmund's code and made some additions to translate the code in 3D.

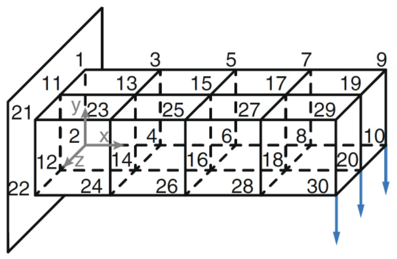


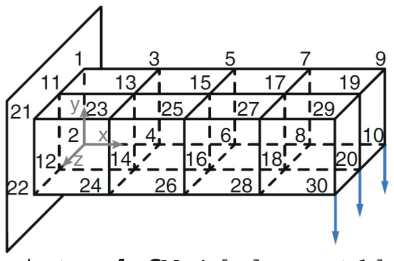
Figure 26: *Node numbering*

The first important thing that forms the basis of an accurate translation, is the numbering of the nodes and DoF's. Liu and Tovar propose the node ID's as shown in figure 24. When the numbering is right and edited all throughout the algorithm, not many problems should arise. This is, as Liu and Tovar showed in their paper, because the structure of the

The input is identical to the 2D, only that `nelz` is added. The supports and forces are placed on the DoF's on the same way. The node ID's are arranged in an array called **`edofMat`**, which shows the DoF's per element. As each element has 24 DoF's attached to it, the size of this array is nele x 24. This matrix is a lot harder to write, and is partially hardcoded in the **`repeatingMatrix`** matrix. This is done by the following lines, translated from Matlab into Python:

```python
#create proper IDs
nodegrd = np.reshape(np.arange(0,(nely+1)*(nelx+1),1), [nelx+1,nely
    +1]).transpose()
newitems = np.transpose(nodegrd[:-1])[:-1].transpose()
nodeids = np.reshape(newitems,[nely*nelx,1])
nodeidz = np.arange(0,(nelz-1)*(nely+1)*(nelx+1)+1,(nely+1)*
    (nelx+1))

nodeids = np.repeat(nodeids, len(nodeidz), axis=1) +
    np.repeat(nodeidz,len(nodeids), axis=0).reshape(nelx,nelx*nely)
    .transpose()

edofVec = (3 * np.reshape(np.transpose(nodeids),-1) + 4)

repeatedVec = np.repeat(edofVec, 24, axis=0)
    .reshape(len(edofVec),24)
repeatingMatrix = [
    0,1,2,(3*nely+3),
    (3*nely+4),(3*nely+5),(3*nely+0),(3*nely+1),
    (3*nely+2),-3,-2,-1,
    (3*(nely+1)*(nelx+1)+0),(3*(nely+1)*(nelx+1)+1),(3*(nely+1)*(
    nelx+1)+2),(3*(nely+1)*(nelx+1)+3*nely + 3),
    (3*(nely+1)*(nelx+1)+3*nely + 4),(3*(nely+1)*(nelx+1)+3*nely +
    5),(3*(nely+1)*(nelx+1)+3*nely + 0),(3*(nely+1)*(nelx+1)+3*nely
     + 1),
    (3*(nely+1)*(nelx+1)+3*nely + 2),(3*(nely+1)*(nelx+1)-3),(3*(
    nely+1)*(nelx+1)-2),(3*(nely+1)*(nelx+1)-1)
    ]

repeatedMat = np.repeat(repeatingMatrix, nele, axis=0)
    .reshape(len(repeatingMatrix),nele).transpose()
edofMat = np.add(repeatedMat, repeatedVec)
```

Pythoncode 8: *Construction of the edofMat array (Liu & Tovar, 2014)*



```
print edofMat[element1]

[3 4 5 9 10 11 6 7 8 0 1
2 32 33 34 38 39 40 35 36
37 29 30 31]
```

Figure 27: *Example of edofMat*

The main reason that this array is created, is to assemble the global stiffness matrix, **K**. Just as in 2D, the K matrix is constructed by creating the row and column indexes and a separate matrix, sK, for the element stiffness matrices. Sigmund's code needed for-loops to retrieve the rows and columns, Liu and Tovar avoid these loops with the Kronecker product, which saves quite some time (Liu and Tovar, 2014). This is also already applied in pseudocode 8. Eventually the matrix **K** is created as a sparse matrix, as a lot of values in the matrix are 0's, especially when the translation to 3D is made. The size and shape of this matrix is important for the efficiency of the algorithm, but chapter 3.8 will go more in depth about efficiency.

In the 3D problem, solving the system follows the same rules as the 2D problem, so that the force-vector and the K-matrix have the same length. This will result in a displacement-vector, which is calculated for each DoF.

Translating these displacements into the sensitivities is identical as in 2D, as K is an 2D array and both U and f are 1D. It is important to note that the edofMat vector was designed for Matlab, so therefore all the values have to be subtracted by 1. The following lines define the objective and its sensitivities, which are identical to pseudocode 7:

```
edofMat = edofMat - 1
#Objective and sensitivity
ce = (U[edofMat].transform * KE) * U[edofMat]
ce = sumof(ce, axis=horizontal)
obj = sumof( $x^p$ * ce)
dc = - p * $x^{p-1}$ * ce
```

Pseudocode 9: *Calculating the obj (compliance) and dc (sensitivities) for 3D*

Lastly, the values are updated using the optimizer to give new values of x and get into the next loop. Overall, the 3D approach doesn't change very much from the 2D approach, the biggest change is the indexing of the nodes and having a proper administration of these. When the translation of the design space to the matrices can be made, the calculations can be made, and the matrices can be translated back to geometry.

>>>3.2.4 Alterations of these codes

In his paper, Sigmund already wrote several other options and alterations that can be implemented in his script. Of course, his goal was to let his code work with as many problems as possible, therefore the forces and supports are created as a user input. Sigmund shows different possible force and support options to solve corresponding problems. Sigmund does also elaborate on the addition of multiple forces. When applying multiple loads on the system, Sigmund argues that in this case the F-vector becomes a vector with n columns, where n is the amount of loads. The objective becomes the sum of the compliances, as the resulting compliance will be a vector with n columns as well. It has to be noted that later versions of Sigmund's code (written by Andreassen and Deetman) already implement this in their code.
A second function that Sigmund added was the implementation of active and passive elements. In the problem sometimes several elements should never exist, or should always exist. Sigmund added a passive array with the size of nelx x nely which includes 1's at places that should be passive. After the x is updated, as shown in listing 12, the elements following the passive array, are set to 0.001. This ensures the element will always be a void. The opposite can be done as well, with a separate active array. The following lines are from the Matlab code, written by Sigmund:

```
voidlist = [void1, void2, .... voidn]
voids = zeros(size=nele)
voids[voidlist] = 1
....
x = where(voids = 1, x = 0.001, else x = x)
```

Pseudocode 10: *Implementing voids*

>>>3.2.5 Creation of the Python code

In order to understand each line better, the code that COMPAS used was first thoroughly examined, line-by-line. This made sure each step of the algorithm was properly understood and adjustments could be made easier. Due to the implementation of MMA, in the rest of the process the code, written by Deetman, was used as the basis. As the algorithm will work within Grasshopper, the main definition is transformed into a linear algorithm, with the initialization of the variables as a first step. Also some lines are added in order to quickly add voids, forces and supports from Grasshopper. In order to show the calculations, matplotlib is used to plot the results.

The algorithm include both OC and MMA, where OC is written in a definition on its own, and is only called when `xsolv` is set to "OC Method". Otherwise the MMA algorithm is used, which uses an imported library. This library is also written by Deetman, a Python version of Svanbergs MMA-code. Two definitions are used from this library, mmasub and subsolve. As importing libraries into Grasshopper can be quite hard, these subroutines are added as a definition on the bottom of the script. This way, only Numpy has to be added from GhPython.

>>>3.3 Methodology

This research is focused on finding topology optimization methodologies that are applicable in architectural problems. The objective was stated in chapter 1.3 as to *implement density dependent forces in topology optimization and apply this algorithm to buildings*. To achieve this objective, several sub-objectives were defined, in order to achieve this objective. The first objective, to *create and understand a topology optimization algorithm and translate this in Python*, has already been spoken of in chapters 2, 3.1 and 3.2.

For the following objectives the following chapter will give a detailed methodology, based on several case studies. Each objective is cut into two practical examples that need to be solved. Each problem is chosen in a way, so that it tackles a few problems at the time and works towards solving the individual sub-objective. As there are three sub-objectives, six toy problems are developed and later on solved. The following chapter will discuss these problems, and what the problems/objectives of each one is.

To ensure a systematic approach, each problem is individually looked at following the methodology as proposed in chapter 1.6. The process can be summarized as (Peffers et. al., 2007):

- Build
- Evaluate
- Theorize
- Justify

One step before this process is the problem identification and motivation, which is performed in this chapter.

>>>3.3.2 Setting up an algorithm

The first sub-objective is to *"create and understand a topology optimization algorithm and translate this in Python"*. To achieve this, two toy problems are created and initiated in this chapter.

TOY1: The cyclist tunnel
The first toy problem is shown in figure 28 and represents a section of a tunnel, that is used for cyclists. This problem is a standard problem that has not many variables or difficulties. The main objective for this toy problem is to get the algorithm running and trying to apply some basic constraints and solve it with both OC and MMA.

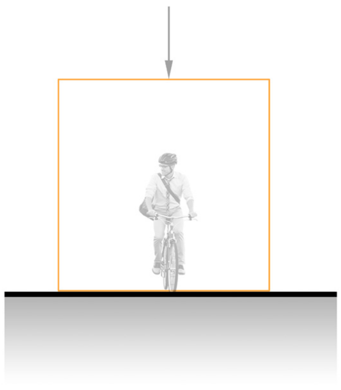


Figure 28: *TOY Problem 1*

As shown in figure 28, there is a single force on the roof of the tunnel, pointing downwards. The full ground can be used as a support, except for the voids. Adding the void for the cyclist to pass is important to check if voids work properly and if it is processed successfully in the loop. As the inputs don't really change that much in other toy problems, this problem will also process the Grasshopper inputs. Later on, this code is translated to a Grasshopper plugin, but this is discussed in chapter 3.8.

TOY2: The building
The second toy problem is a section of a building, with several rooms as shown in figure 29. The goal of this problem is to further extend the possibilities of the algorithm, and add some new possibilities. A more complex building is chosen, as this raises some new problems.

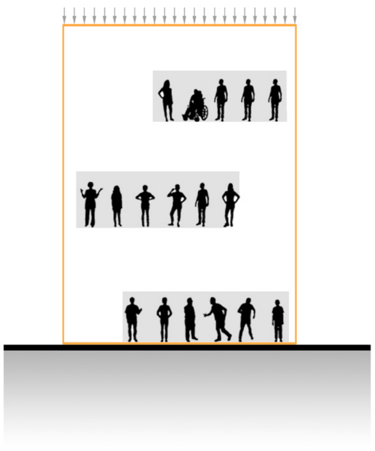


Figure 29: *TOY problem 2*

Support stay the same, as the ground is chosen (or at least a foundation that lies beneath it). The building has three floors and there are three voids. Properly handling these voids and generating these within Grasshopper is important for this problem. This script should also be able to deal with more complex design spaces. Another thing that changes significantly is the way forces are distributed on the roof. In architecture it's very unlikely a single force exists in the design space, but usually a distributed load lies on the roof in the form of snow. Also, all the voids are now rooms and therefore also have their own force on the lowest pixel of the void. Implementing distributed loads and a load in the middle of the design space are the objective to solve this Toy problem.

>>>3.3.3 Implementing density dependent loads

The second sub-objective is *to implement density dependent forces in the methodology and in the algorithm*. Where the first two toy problem were mostly about implementing code that was written by others, in this chapter new code has to be written.

TOY3: A small earthy house

The third toy problem focusses on implementing selfweight on a very small and simple house. This house is made of clay and therefore the building has a large weight that is working like a force downward. The algorithm will need a point to start with, there needs to be a force on which the selfweight can react. Again, the ground counts as the supports with the exclusion of the voids.

It is important to look at the behavior of the self-weight and what it does to the design. Questions about the weight of the self-weight should be asked, as well as the placement of these forces. The house is in 2D, so no doors or windows are added yet. It is expected that the section should look like the picture of figure 31, which is a dynamic relaxed building.

TOY4: An igloo

An igloo has to withstand large portions of snow on its roof, while also supporting its own weight. This is also true with the small earthy house, as there still is a force on the roof. This force is not set on a place, but just on the highest point of the roof. This also has influence on the shape of the design and will work like another density dependent force.

Another piece that has to be looked at is the igloo needs to be covered, to maintain heat inside. There can be no holes in the roof. The optimizer should only consider solutions that have a full roof over the voids. As MMA is used, constraints can be implemented, which is what needs to added to the algorithm in this toy problem.

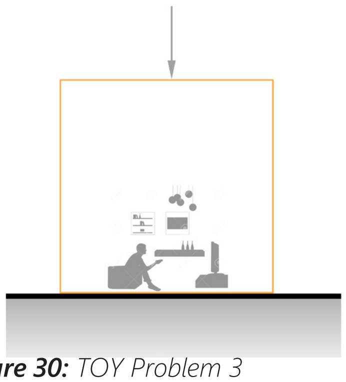


Figure 30: *TOY Problem 3*

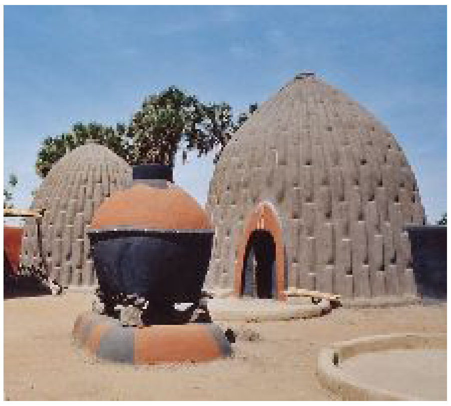


Figure 31: *Dynamic relaxed building*

Figure 32: *TOY problem 4*

>>>3.3.4 Translating into 3D

The third sub-objective is to *generalize the methodology and apply the algorithm to the 3D design spaces and implement the previously written toy problems*. This will partially done by the paper and code written by Liu and Tover and taking their translation as a reference.

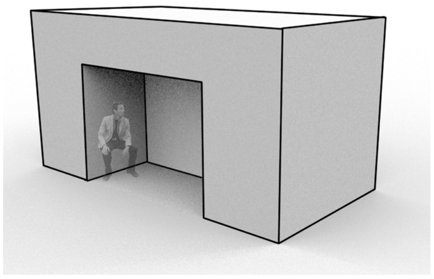


Figure 33: *TOY Problem 5*

TOY5: The bus station
Toy problem 5 focusses on creating geometry in the 3D field with topology optimization. A small bus station is created with a small waiting area for people to wait in. A bus station is chosen because it needs to have an open wall for passengers to enter the bus. This is a problem that is specific for 3D problems, as a section of this would not be able to generate in 2D. There is a distributed load on the roof (so that the roof is flat) and again, the ground counts as a support. In this toy problem it is important to create a working indexing system, that properly handles the voids.

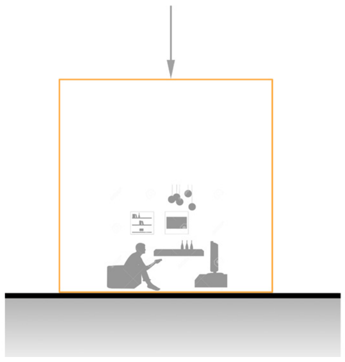


Figure 34: *TOY problem 6*

TOY6: A more complex house
The last toy problem combines all the previous toy problems together, in order to form a small house. The house is also made of clay and self-weight needs to be taken into account. No forces are placed on the design space, the algorithm should make the full geometry itself. No holes may appear in the roof. Lastly, windows and doors should be placed in the design as well, in order for the voids to be accessible. Whenever this works properly, variants of this toy problem can be created, for instance to access multiple floors, in order to research more into its architectural properties.

>>>3.5 Solving the toy problems

This chapter will follow all the steps into developing (building) the algorithm so that it functions as the description in chapter 3.3 is met. Note that this is a process and the toy problems are merely the smaller steps to achieve the main objective.

>>>3.5.1 Toy problem 1: The cyclist tunnel

The objective of this toy problem is to get the algorithm that Deetman wrote, running in Grasshopper and make sure we can edit the inputs. Chapter 3.2.4 already showed some changes that were made into the code in order to get it to run in Grasshopper.

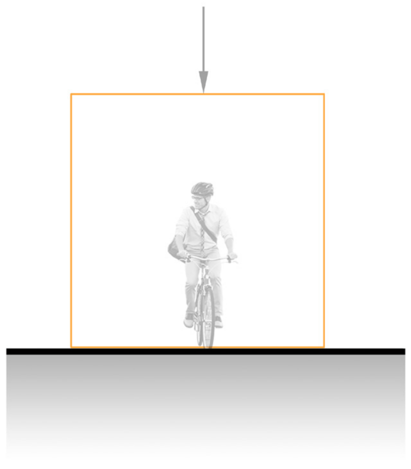


Figure 35: *Toy problem 6*

This script cannot function well in Grasshopper, because running this requires NumPy. This is a library that is not included in the Python that Grasshopper works with. Grasshopper uses IronPython, in which you can't install other libraries like NumPy. There are only a very few ways to get around this problem and the most common one is to use a Proxy server. With these server NumPy is accessed from outside Grasshopper, so that packets of NumPy can be used. This is essential, as the NumPy library contains a lot of matrix transformations that are needed in topology optimization.

COMPAS is a library that is made for GhPython and includes such a Proxy server. A Proxy server is started by the following lines (COMPAS, 2020):

```
from compas.rpc import Proxy
###As numpy is needed, we build a server that access numpy from
    outside GHPython
np = Proxy('numpy')
array = np.zeros(10,1)
```

Pseudocode 11: *Accessing numpy in Grasshopper*

This starts a local server that accesses NumPy for the duration of the script. When NumPy is only used once, XFunc is another function in COMPAS that does something similar as Proxy (COMPAS, 2020). But in stead of starting the server for the duration of the script, it starts it for each line. Therefore, Proxy is much quicker and easier to setup.

Now that the script is working the toy problem can be solved. In order to solve the problem, first the inputs have to specified. To solve this specific problem the following values for the variables are chosen:

topop(nelx,nely,volfrac,penal,rmin) with *topop(100,100,0.3,1.5,3.0)*

This ensures that the design space will have 100 x 100 elements in the end result and this should give an accurate view of the generated design. The other values are standard values that are described in chapter 2. Another way to determine the elements is to draw a surface in Rhino and divide it into elements. This way, specific design spaces can be imported and a `VoxelSize` can be chosen.

The other needed inputs are the forces, supports and voids, which will be retrieved from Grasshopper. This allows for easy initialization and a way to play around with the algorithm. The nodes are placed in Grasshopper from the center point (0,0) and into a rectangle until the coordinates of *(*`nelx` *+ 1,* `nely` *+ 1)*. It is important to note that ID of the node at (0,0) is not 0, but has the ID of (`nely` *+ 1)*. This has to be taken into account when doing any translation from and to Rhino geometry.

To translate the forces to a vector, the user can take Rhino points and use these as an input. The coordinates of the point can be retrieved and translated in to the corresponding DoF, on which the force should be placed. Another piece of information is the direction of the force, this is performed by a vector with a True value when it is in this direction. A [1, 0] vector means vertical and a [0, 1] means horizontal. The same principle goes for individual support points. Figure 36 shows an example of a simple cantilevered beam that is created with supporting points and a point load. Another component is created in order to show where loads and supports are. This model is very important in later plugin development, chapter 3.6 will further go into this subject.

Listing 14 also contains a simple way of showing the final generated geometry. The output of the algorithm, x_s can be translated in a True and False list, which is True for each value of x over 0,3. When culling the initial geometry with this list, the generated geometry is received.

The toy problem states that the ground should be the full support of this tunnel, and foundations are made wherever needed. To do this, inside the algorithm the following lines are added:

```
#Setting floor as support
allDoFs = arange(0::nDoFs)
fixed = emptyList
for element in range(0:nelx + 1):
      nodeID = element * (nely + 1) + nely
      fixed.addtolist(2*nodeID)
      fixed.addtolist(2*nodeID+1)
freeDoFs = allDoFs.subtract(fixed)
```
Pseudocode 12: *Configuring the ground as supports*

Each `nodeID` with the y-coordinate equal to `nely`*+1* should be a support. This is added as a possibility, so when the variable `suppGround` is equal to *True*, this is activated.

The last thing that has to be added are the voids, which should be created in Rhino. To do this, the design space is drawn as a surface and the outline of the void is drawn as a curve. The library Rhino.Geometry contains *Curve.Contains(point)*, which returns True when the point is inside the curve. The following lines are added around the element creation from the lines at pseudocode 13.

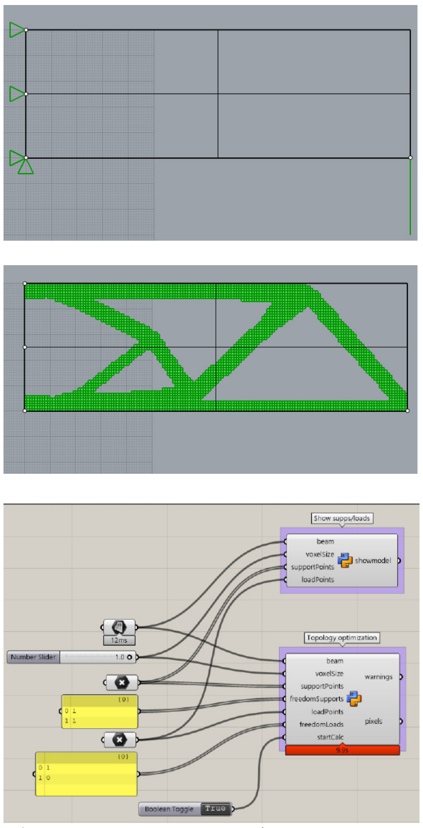


Figure 36: *TO in Grasshopper*

Two arrays are generated, one with the `nodeIDs` of all the elements, and one with the `nodeIDs` of the voids. This second list works somewhat the same as Sigmund's passive list, as shown in chapter 3.2. It is important to also cull the ID's at the supports, as in the FEA these still count as supports. This is done with the lines:

```
insideList = emptyList
for element in allElements:
    currentPoint = rg.Point3D(xvalue, yvalue, 0)
    isInside = curve.Contains(currentPoint)
    if isInside == Inside:
        insideList.append(indexofPoint)
```

Pseudocode 12: *Get points inside curve*

When all the elements above are added and the right inputs are given the following is the result. This is calculated using optimality criteria and shows the created geometry. It can be seen that the void is properly neglected and that around the force the geometry is present. It seems that it has properly transformed into an optimized geom

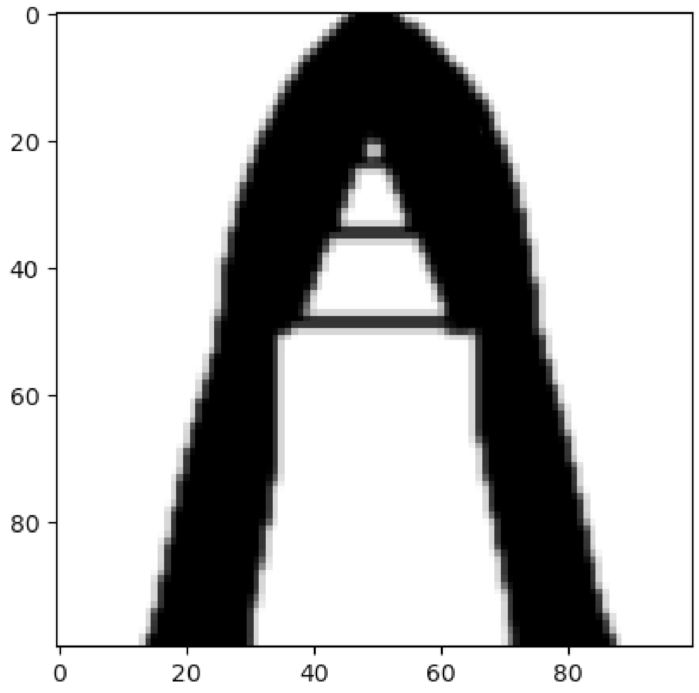


Figure 36: Result of toy problem 1

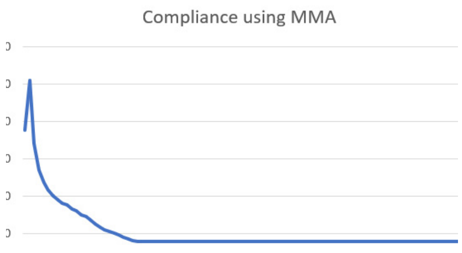


Figure 37: Compliance over the iterations

The algorithm took around 4 minutes (using MMA) to get into the needed equilibrium (100 loops) and the graph at figure 30 shows how the compliance behaved in this process. As expected, this happens at a quick rate early on but slowly gets into very little differences. Further optimization and calculation time results are found in chapter 3.5.

Lastly the models will be compared with another piece of software, in order to see if they are giving feasible results. The comparison is made with Ansys, an extensive program that includes static structural calculations, as well as topology optimization. As only the student license is available, the number of elements in this model is restricted. Because of this reason, the models can be a little inaccurate, but overall provide good reason of validating the results.

In order to solve these problems in Ansys, first the engineering data have to be initiated. In the toy problems always E-values of 1 are chosen and the same goes for variables like the volume and forces. Then geometry is drawn using SpaceClaim that corrospond to the design problem. In 2D problems, the geometry is drawn in the XY plane and no geometry is drawn in the Z direction. Then, the model is created by first meshing the geometry. The mesh is linear divided in a way that squares are created. If the license allows it, the mesh is divided in the same number of squares as there are number of elements in the problem. Eventually this gives the following results:

The first toy problem is shown in figure 37 and shows the result from Ansys, using the same geometry. A square is created with a hole for the cyclepath. The geometries are very similar and show the same behaviour. An A-shape is created with a hole above the design space. The ground shows a bit different, which is possible because of how Ansys defines the loads. Because the full surfaces of the design space are set as supports, the elements connected to these surfaces are automaticly rendered. The algorithm that is created does not do that, which results in this difference.

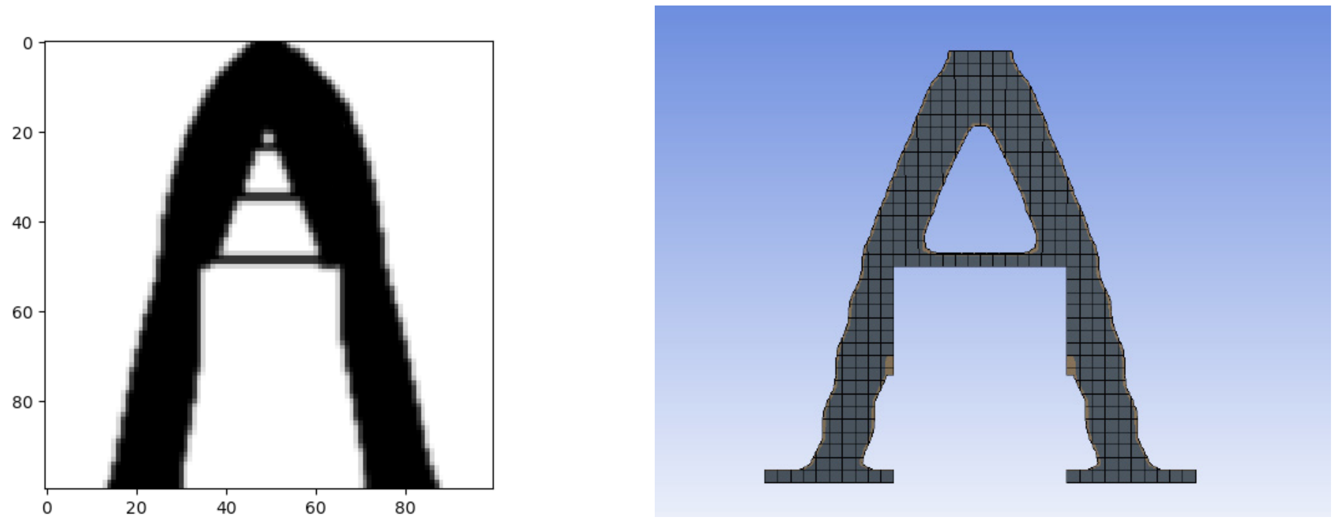


Figure 38: *Comparison with the results (left) with a calculation made by ANSYS (right)*

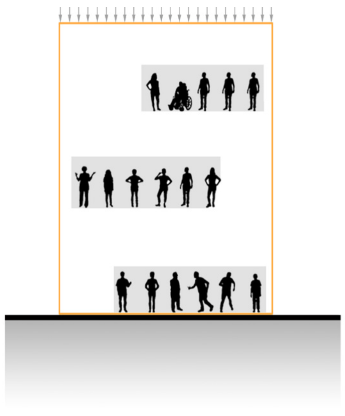


Figure 39: Toy problem 2

>>>3.5.2 Toy problem 2: The building

The second toy problem focusses on more complicate versions of the script that is created in 3.4.1. There are four main objectives that need to be solved in this toy problem. Area loads, more voids, complex design spaces and forces inside the design space. Implementing these, allow for more sections that make sense in architecture.

The methodology that was proposed in 3.4.1 for the voids seems to work as well for multiple voids. The only change that has to be made is that for each curve that is given as an input, all the points have to be checked. A very slight change to the line below has made, in order to ensure the possibility of multiple outlines.

```
insideList = emptyList
for element in allElements:
      currentPoint = rg.Point3D(xvalue, yvalue, 0)
      isInside = listOfCurves.Contains(currentPoint)
      if isInside == Inside:
            insideList.add(indexofPoint)
```

Pseudocode 13: Get point in any curve

Complex design spaces are possible in this algorithm as well and require a smart way of dealing with this. The method that is used is to take the design space as a surface. The outer edge of this surface is treated like a curve and divided in points. The point with the highest x-coordinate and y-coordinate is chosen, and a rectangle is created with the size x + 1 and y + 1. This ensures the design space is fully in the new rectangle. Then, this rectangle is divided in elements, where elements outside the design space are culled. Another method is not viable, as the element indexes need to be in a form that matrices understand. Also, this is in the initializing phase and should not add to computational time.

```
points = surface.outline.divideInPoints
x, y = points.getPointWithHighestCoordinates[x, y, 0]
designSpace = rectangle(shape=x+1, y+1)
elements = designSpace.divideInPixels(x+1, y+1)
for each element in elements:
      surface.outline.Contains(element)
      if isOutside == Outside:
            voidList.add(element)
```

Pseudocode 14: Implement complex design spaces

Next area loads are added. A similar approach as with the ground indexing is used, where all the points are selected with the y-coordinate is equal to 0. It has to be noted that the first and the last node only get half the force. Also the force on each DoF should depend on the chosen size of the force, divided over all the elements. This can be done with the following lines to define the force:

```
forceValue = 0.01
for roofelement in range(nelx + 1):
      nodeLeft = roofelement * (2* (nely+1)) + 1
      nodeRight = (roofelement+1) * (2* (nely+1))+1
      f[nodeLeft, nodeRight] = forceValue / 2
```

Pseudocode 15: *Area loads on a roof*

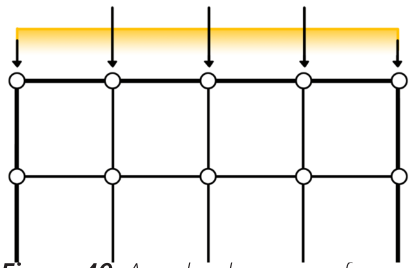


Figure 40: *Area loads on a roof*

Impementing the area load and the voids results in the geometry shown in figure 41. Lastly, placing forces inside the design space behave the same as area loads, but instead on them being on the roof, they exist on the element that is directly beneath a void. Each void can be assumed to be some kind of room and will deliver a force on the floor. The way to place these forces is to check all the elements that are voids and see if the next one (when counting from the bottom down) is a void as well. When that is not the case, the next element is always the floor of that void. An force is placed on each of the DoF's corresponding to that element.

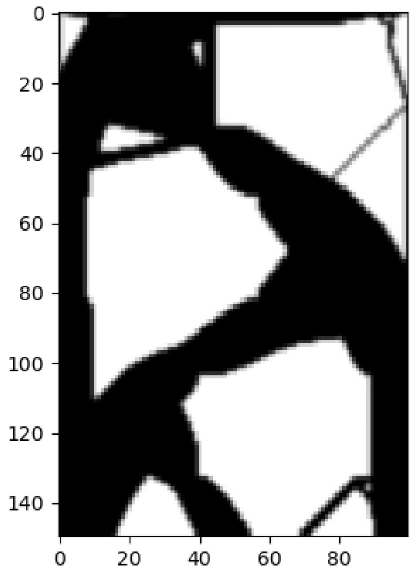


Figure 41: *Area load and voids in toy problem 2*

```
#Voidlist contains indexes of the voids
forceValue = 0.01
for void in voidslist:
     if void + 1 is not in voidlist:
           nodetopleft = xvalue * 2*(nely+1)
                                + 2 * yvalue +1
           nodetopright = (xvalue+1) * 2*(nely+1)
                                + 2 * yvalue + 1
           f[nodetopleft,nodetopright]= forceValue/ 2
```

Pseudocode 16: *Implementation of floor forces*

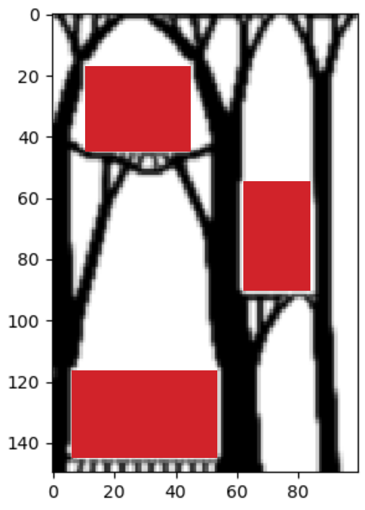


Figure 42: Voids in the system

All these changes are applied to a new design space, that follows the shape of the building as shown in figure 39. The voxelSize was increased to 2, in order to save some computational size. The design space that was created had a nelx of 100 and a nely of 150. All the voids are created in their own list and the forces are applied on their floors, as well as on the roof. Figure 42 shows the voids in the system and how they are placed. Finally, the calculation was made and the result is shown in figure 43.

The final geometry clearly ignores the voids and no voids are filled. The roof clearly has an area force downwards and so do the voids. The final result has a clear vertical direction, which makes sense as there are only vertical loads. Especially the right voids starts to look like a section of an old church, with its high arch. Another architectural principle that is created is that of hollow core floors, especially on the bottom most void. In stead of filling in the area, holes are made in the floor in order to save material.

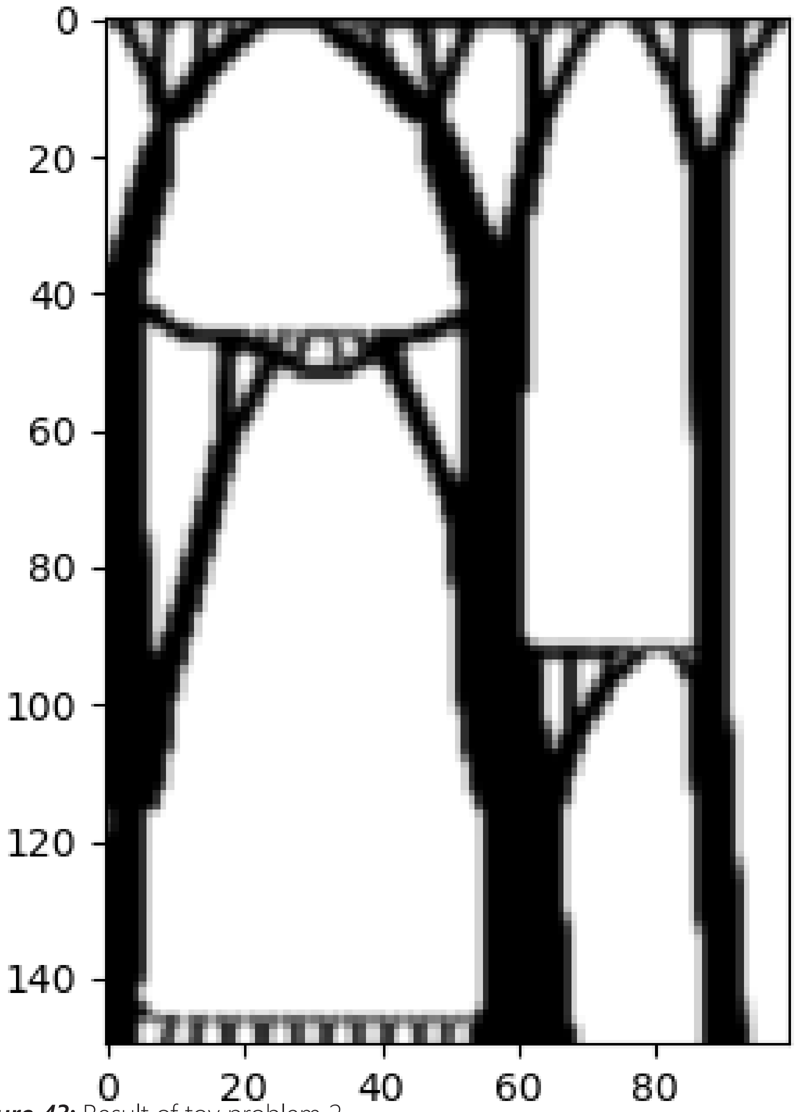


Figure 43: Result of toy problem 2

The second toy problem is shown in figure 42 and will be compared with the result of an identical design space in Ansys. In Ansys, a design space is created in the XY plane with identical dimensions and voids. The geometry is meshed into squares, with as much faces as the student license allows to have. The result of the Ansys optimization can be seen in figure 44.

Figure 44 shows that both geometries clearly represent the same solution. Both geometries have high arches around the voids and very similar shapes. The geometry that was generated in Python seems tho have rounder arches above two of the voids. The main difference is to be seen in the left corner, where the algorithm removes material, while Ansys doesn't. This probably has to do with the way Ansys registeres supports, where a voxel that is registered as support always gets material. Interesting is also that both geometries include the hollow core floors, which is remarkable.

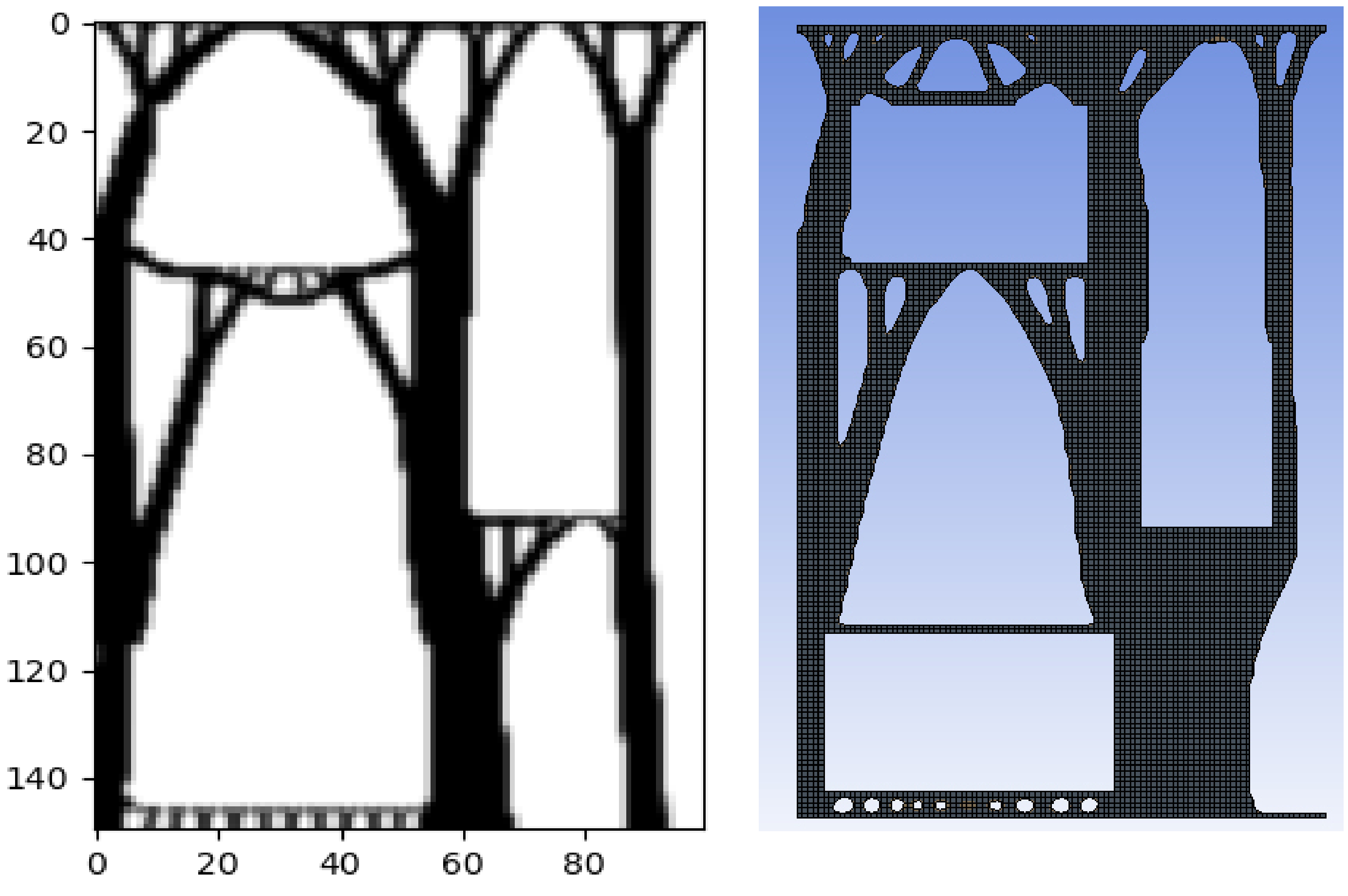


Figure 44: Comparison toy problem 2 with ANSYS

>>>3.5.3 A small earthy house

The house exists of one space with very few constraints, there are no windows or doors. The only constraints that are important is that the ground counts as a support, there is self-weight and the void should be closed off at the top. Implementing self-weight is the main issue of this toy problem and should be fixed in this chapter.

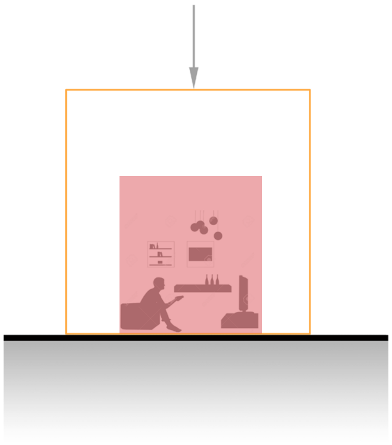


Figure 45: *Toy Problem 3*

Chapter 2.6 explained why selfweight is quite hard to implement, because a new variable arises. Placing an element will no longer influence the stiffness positively, but can also increase the compliance. A few things have to happen in order for this to work and this is shown by a simple example of this design space. Figure 45 shows the design space without self-weight. Self-weight can be defined as a force that is placed on each element that exists, shown in figure 46. Or in other words, the value of x multiplied by some factor. This is described by Bruyneel & Duysinx as the following (Bruyneel & Duysinx, 2001):

$$f_{i,j} = x_{i,j} \; a_g \; V_i \; / \; 4 \qquad [19]$$

Where the bodyforce f is equal to the density of an element, multiplied by a factor a, which determines the strength of the selfweight. This should be dependent on the total volume. This is multiplied by the volume of the element V, which in this case is equal to 1. This force is divided by the 4 nodes and placed on the vertical DoF. In this research the force on each node is defined as:

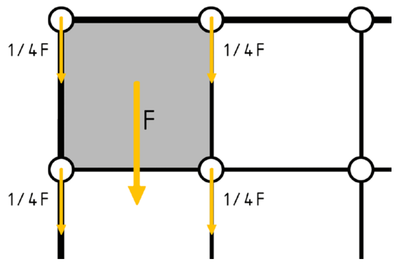


Figure 46: *Selfweight on an element*

$$f_{i,j} = x_{i,j} \; selfweight \; / \; 4 \qquad [20]$$

To determine the selfweight forces, a *selfweight* value is calculated, based on the function that Bruyneel and Duysinx created. This selfweightfactor *a* is divided over the nodes that are corrosponding to the vertical DoF's. These vertical DoF's can be retrieved from the matrix **`edofMat`**, created in pseudocode 3. From this matrix, only the even indices are taken, to only take the vertical DoF's, creating the matrix **`yVertedOf`**.

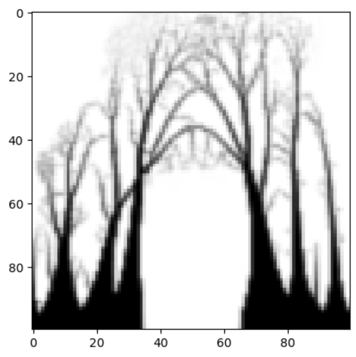


Figure 47: *Selfweight without a threshold*

Figure 47 shows the result when applying the selfweight directly on the algorithm, without a threshold. In iteration 1 all the values are given the value *volfrac*, and will generate selfweight. These selfweights will combine and create this a result that looks more like a forest. To counter this, values that are closer to 0, should be penalised in order to not to gain selfweight, where values closer to 1 should create selfweight.

Introducing the selfweight in equation [21], the density is powered with a penalisation power, $x_{self.}$

$$f_{i,j} = x_{i,j}^{\;p} \; selfweight \; / \; 4 \qquad [21]$$

Several calculations show that even with a very high p, the result is not ideal, and the tree effect still happens. This is due to the fact that the optimizer will try to set the value of the density so, that the selfweight is minimized. It will choose the densities to be grey, in stead of black and white. Even with a p = 99, the optimizer will always try to find grey densities, as this will significantly lower the compliance. A solution was found in the use of an heaviside smooth function (Huang & Deng, 2018):

$$x_{rounded} = \frac{1}{2}\frac{1}{\pi}arctan(\frac{x-a}{s}) \quad [22]$$

This function returns a 1 for all the values that are larger than a, while returning a 0 for all the values smaller than this mark. Calculating this value for all the values of x will return a rounded value that is either 0 or 1. When multiplying this with the selfweight value, the selfweight can be calculated. In this function, the value of s is the slope, in which a smaller number equals a more steep slope. But just as the power function, putting this into the sensitivity is not preffered, as the optimizer will choose grey values.

When the forces are applied to the DoF's, the sensitivity has to be adjusted. Chapter 2.6 showed a new sensitivity in order to solve self-weight problems. This sensitivity is now:

$$C = \sum_{nele} x_e - U_e^T K_e U_e - 2U_e^T F_e \quad [17]$$

This new sensitivity exists of the displacement vector and the derivative of the force, that is caused by the selfweight. Eq. [20] shows that this force is the density, multiplied by some constant factors. As we need to find the derivative of F over *x* and the smooth step function is not taken in the sensitivity, the derivative of [17] is equal to:

$$\frac{C_e}{x_e} = -px_e^{p-1} - U_e^T\frac{\delta Ke}{\delta x_e}U_e - 2U_e^T\frac{\delta Fe}{\delta x_e} \quad [18]$$

Where dF/d*x* is a vector with selfweight / 4 on all the vertical DoF-indexes, as selfweight can only occur in these vertical DoFs. This is very easy to create using the following line, where swfactor is a user input:

```
selfweightvector[1::2] = swfactor/(volfrac * n)/4
```

Pseudocode 17: *Derivative of selfweight*

Lastly it should be important to note that MMA has to be used when solving self-weight problems. OC can not handle non-monotonous functions and is therefore not usable. Looking at the vector that is created at pseudocode 17 it can easily been seen that result of a multiplication with the displacement vector can result in positive values. Therefore the whole senstivity can result to be positive.

In the algorithm the calculation of the selfweight is performed in the beginning of the main loop. It forms the basis of the force vector, where any preset forces are added right after. Pseudocode 18 shows this and also shows that it is necessary to reset the force vector each iteration, as all the values change each iteration. Any preset forces will also be added in this position in the iteration.

```
 selfweightvector[unevenIndexes] = swFactor / (volfrac*n*4)
start iteration:
      f[all] = 0
      xrounded = smooth_step(x)
      f[verticalDoFs] = xrounded * selfweightvector
      f[presetIndex] = valuesOfF

      U = solve(K,f)

      optimize the system
```

Pseudocode 18: *Implementation of selfweight in the total algorithm*

An important note that has to be made is that in this research precise and validated results are not calculated. This research is mainly focussing on shape generation and could give a basis for these results. It is important to balance the factors of the selfweight, there should be a logical relation between all the forces. By default these forces are now put on balances values, chapter 3.9 will futher discuss how to handle these factors and user inputs.

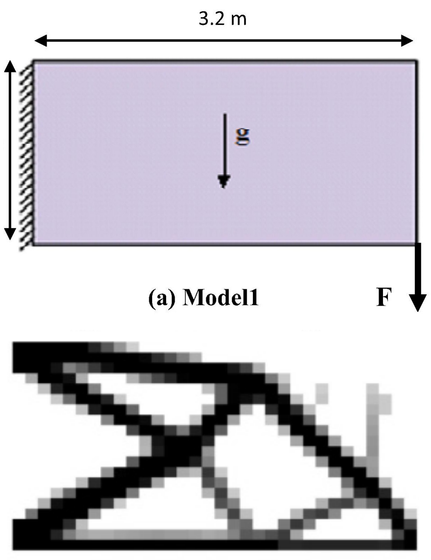


Figure 48: *Conditions and result of the beam by (Jain & Saxena, 2018)*

Results

This page contains some of the results that are created with the implementation of selfweight in the algorithm. It is important to first check if the algorithm works with different amounts of elements, volfracs and placement of loads and supports.

Figure 48 shows a standard beam that was introduced in toy problem 1, which is a cantilevered beam with a force going downwards. Jain & Saxena preformed topology optimization on this beam and their results will be taken as validation to check if the result is believable. In their research, the effect of selfweight on topology optimization is researched, mainly towards the size of this selfweight (Jain & Saxena, 2018).

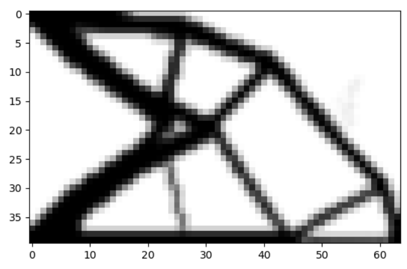


Figure 49: *Result of the same beam in this algorithm*

Comparing the result for the cantilevered beam, the result that was gotten shows great similarity with the result of Jain and Saxena. Both show a V in between the supports and similar shapes towards the force. Small differences are probably caused by the difference of the amount of elements. Also the beam, shown in figure 47, is compared with Jain and Saxena, and shows similarities. The shape is very similar and shows identical properties. A central, thicker, V-shape is generated which is the basis of the beam. The outer geometry is thicker as well, with thinner trusses in the middle.

Figure 50 shows the result for this toy problem. It is fully placed on the ground as was the case with this toy problem. The force is placed on DoF [10363] (`presetIndex`), which is right above the void. The ratio between the force and the selfweight is 1:4, or when looked at the pseudocode the `swFactor` is 4 and the `valuesOfF` is 1.

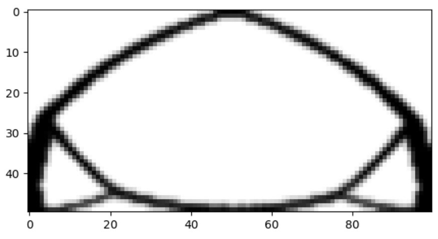


Figure 50: *Selfweight in a beam*

Figure 52 shows the result of this calculation, using these variables. The result looks a lot like a catenary curve that divides the force over the possible supports. This makes sense, as this probably results in the lowest amount of stresses. Towards the top of the void, the shape seems to follow a catenary curve, or at least a close approximation of this. Above the void, a triangle is created in order to divide the force evenly to both the sides. As the selfweight in this triangle is way lower than the force F, it looks like the selfweight has much less influence, resulting in a more straight "cap" on the shape.

The graph in figure 51 shows the compliance over the iterations and shows how the solver handles the non-monotonous problem. The graph follows the shape that was predicted in the paper by Bruyneel and Duysinx, which is also shown in the figure (Bruyneel & Duysinx, 1997). Implementing a more complex system of optimizing the system could increase the speed of the algorithm. The result should not change much, therefore this results is considered as sufficient.

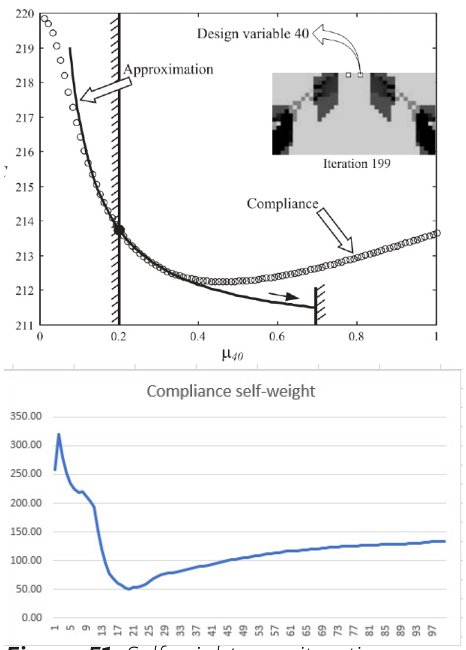


Figure 51: *Selfweight over iterations*

Also an analysis is performed on the shape in ANSYS, calculating von Mises stresses and strains in the system. Note that the shape as shown in figure 53 has been simplified into a single surface and some inaccuracies exist. The result of the calculation shows that the triangle on top is most vulnerable for the forces, which makes sense, as this is not following the catenary curve. The beam in between has most of the stresses and its deformation causes stress in the rest of the geometry. Because of the force, the geometry is not optimal and it would be beneficial to figure out how to generate geometry only based on selfweight.

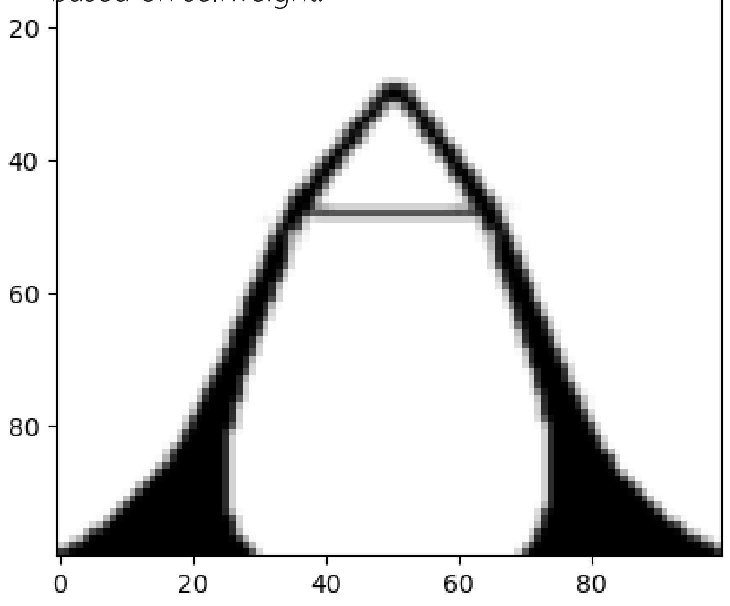


Figure 52: *Result of toy problem 3*

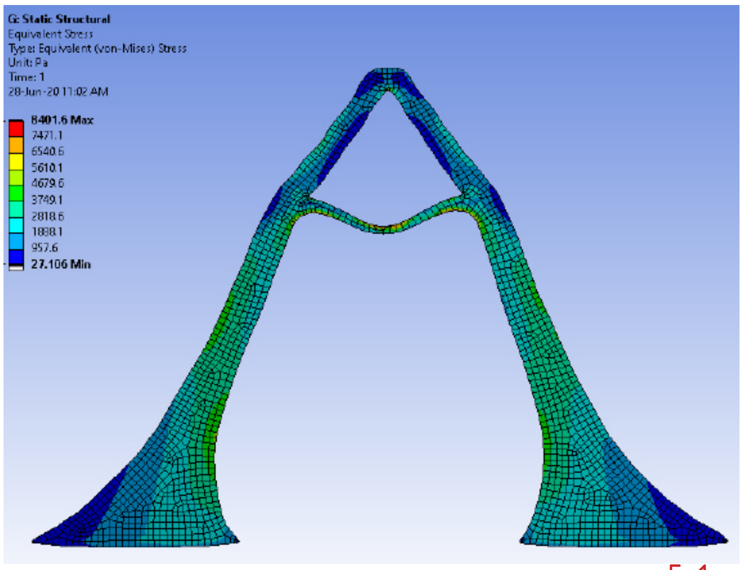


Figure 53: *Ansys analysis of toy problem 3*

>>>3.5.4 *The igloo*

The main problem when only using selfweight is the initial shape. On iteration 0, all the values are 0,3 and will remain that, as there is nothing to solve. When there is 1 force, there is a solution to the system and there is something to optimize. When there is only self-weight this is way harder to solve. In buildings, usually the forces on floors will make sure there are forces to calculate with, but this toy problem doesn't have those. Another constraint has to be used, where the roof could be this constraint. When the void is placed, it has to have a roof over it, otherwise it will not be optimal. This will not change in any iteration, so it is important to keep checking if there are holes in the roof.

As MMA is used, another constraint can be added that the optimizer has to take care of. This constraint is developed following the guidelines of M. Langelaar (Langelaar, 2020). To check if the roof has holes, the sum of all the elements above all the voids should be larger than 1. In that case, it can be assumed it has a roof and this is preferred by the optimizer. This can be rewritten as:

$$\sum(x_{k,i}) \geq 1 \qquad [24]$$

for all columns where a void exists. If the set of columns with a void is written as K, [24] can be rewritten as a constraint function, g:

$$g_k = 1 \sum_{column_k} x_{k,i} \qquad [25]$$

$$g_k \leq 0 \qquad k \in K \qquad [26]$$

Using a n-number of constraints is not preferable for the optimizer, so all the constraints are combined in a KS function. This function combines several constraints and translates them all into one constraint. This means that when all but one constraint are fulfilled, the function will still output the failed constraint. This ensures that all constraints are met and a proper output is given. This KS function looks like the following (Martins & Poon, 2005):

$$KS(x) = \frac{1}{P} ln(\sum_{k \in K} e^{Pg_k}) \leq 0 \qquad [27]$$

Where P is a factor that approximates the accuracy, chosen to be 10. Implementing this constraint in the MMA optimizer is very handy and not hard to do. Until now, the main constraint was the volume constraint, which was given as input `fval` to the optimizer. Now this `fval` becomes a column vector with the volume constraint and the roofconstraint.

To properly implement the roof constraint further, the sensitivity of the constraint, `dfdx`, has to be added. This sensitivity of [24] can be written as:

$$\frac{\delta g_k}{\delta x_{k,i}} = -1 \qquad [28]$$

The sensitivity of [28] can be rewritten as:

$$\frac{\delta\widetilde{g}}{\delta g_k} = \frac{1}{P}\frac{1}{\sum_{k\in K} e^{Pg_k}} P e^{Pg_k} = \frac{e^{Pg_k}}{\sum_{k\in K} e^{Pg_k}} \qquad [29]$$

Combining these 2 sensitivities will give (Langelaar, 2020):

$$\frac{\delta\widetilde{g}}{\delta x_{k,i}} = \frac{\delta\widetilde{g}}{\delta g_k}\frac{\delta g_k}{\delta x_{k,i}} = -\frac{e^{Pg_k}}{\sum_{k\in K} e^{Pg_k}} \qquad [30]$$

$$\frac{\delta\widetilde{g}}{\delta x_{k,i}} = \begin{cases} \textbf{if } \mathbf{k} \in \mathbf{K} : -\frac{e^{Pg_k}}{\sum_{k\in K} e^{Pg_k}} \\ \textbf{if } \mathbf{k} \notin \mathbf{K} : 0 \end{cases} \qquad [31]$$

Equation 29 shows the final sensitivies for all the elements that are in a column with a void in it, as K is a list of all the columns with voids in it. For all the other elements the sensitivity is 0, as there is no void in the column, and therefore no roof has to be placed.

Implementing this sensitivity in the MMA optimizer follows the same procedure where `dfdx` is concenated into a column vector. Getting these senstivities in the algorithm follows the following pseudocode:

```
voidcolumns = columnindex(sum(voidsMatrix)>1)
invertedarea = 1 - voids[voidscolumns]

epgk = e ^ (10 * sumofcolumns(x ^ p))
gcolumns = np.log(sum(epgk))/10

dcroof = epgk / gcolumns
dcrc[voidcolumns] = tile(dcroof, nely) * invertedarea
```

Pseudocode 19: *Calculation of the roofconstrain (gcolumns) and its sensitivity (dcrc)*

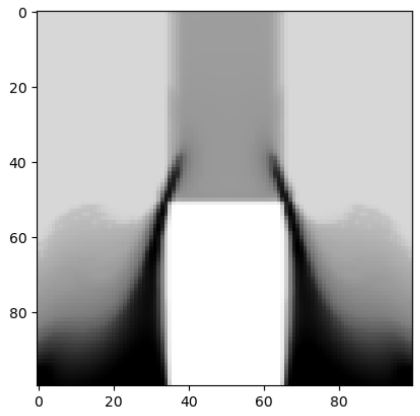


Figure 54: *The roofconstraint working on iteration 11*

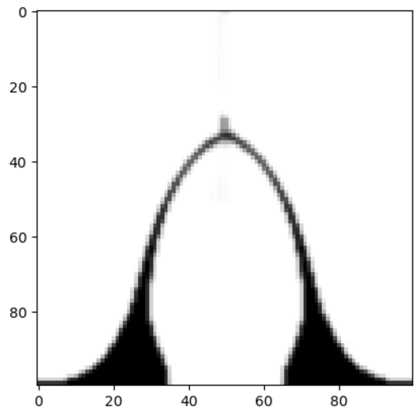


Figure 55: *Final result roofconstraint*

A matrix with 1's at the void-indexes is summed up around its columns. When that sum is at least 1, it means a void exist and its column is added. Then the inverted area is created with a value of 1 above the voids. After dcroof is created, the values are tiled vertically and multplied by the invertedarea, in order to set the value to 0 at the voids. A penalization power has been introduced as well, in order to filter out low values summing up to be more than 1. For clean results a value of p = 5 is chosen.

Figure 54 shows the algorithm running at iteration 11 and shows how the optimizer doesn't see a proper roof and adds values above the voids. A large square above the void is definitely not optimal for the optimizer, so it shrinks it. Finally, the result is shown in figure 55, which is at iteration 498. The only force in this system is the selfweight and a result is retrieved, which was the goal of this implementation. It looks very dynamicly relaxed, but is very thin, as the least amount of material causes the least amount of forces and displacements.

One key element to igloos and buildings overall, is the existence of a snowload. Each building will have a force on its roof, which is dependent on the shape of the roof. When a roof exists, the top voxel should gain an extra force. Or in other words: if the sum of the values above an element, including the element, is equal to 1, then the element will gain a new force. First a new function, y, is created that is the sum of values of a column (Langelaar, 2020):

$$y_{i,j} = \sum_{k=j}^{nele} x_{[i,k]} \tag{31}$$

$x_{i,j}$	$y_{i,j}$
0	0
0.01	0.01
0.1	0.11
0.4	0.51
0.4	0.91
0.5	1.41
0.8	2.21
1	3.21

Figure 56: *Example for snowload*

Now, only values of y that are 1 should return 1, values larger or smaller should return a 0. Values that are nearly 0 or values that are e.g. 0.2 should be neglected (as they are not a roof) and therefore a penalization power will be implemented, just as with the constraint. To also counter this, the load will be divided over elements with y values of 0.5 < y < 1.5. The optimizer will then prefer single elements over multiple elements, which will result in a more black/white roof. Figure 56 shows as system with certain densities, which result in a grey roof. Figure 57 are the y values of the elements, according to equation 31, resulting in 3 elements that have values in the range 0.5 < y < 1.5 and therefore will recieve the snowload.

$SH(y_{i,j})$	dfdx
0	0.011
0	0.011
0	0.077
1	15.9
1	0.011
1	-0.4
0	-0.007
0	-0.007

Figure 57: *Example for snowload*

This is not optimal, so the optimizer will try and reduce the amount of elements that will receive the snowload. To do this the sensitivity of the sum of elements is needed. First a function needs to be found that sets the y values to 1 between 0.5 < y < 1.5.

In order to pursue this two smooth-heaviside functions are used, to get the graph as shown in figure 58. Values from 0 > 0.5 and larger than 1.5 will result in a 0. The function will look like this (Langelaar, 2020):

$$f(y) = SH(y - 0.5) - SH(y - 1.5) \tag{32}$$

SH is the smooth-heaviside function which can be written as (Huang & Deng, 2018):

$$SH(x) = \frac{1}{2} + \frac{1}{\pi}(arctan(\frac{x}{s})) \tag{33}$$

or together:

$$f(y) = \frac{1}{\pi}(arctan(\frac{y-0.5}{s})) - \frac{1}{\pi}(arctan(\frac{y-1.5}{s})) \tag{34}$$

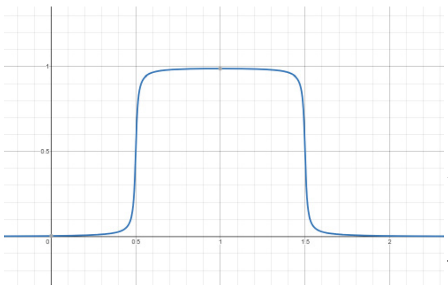


Figure 58: *Smooth step function*

Where s is the slope of the step, which is chosen at 0.01, in order to get a very steep step. This graph is plotted at figure 58. When calculating the example of figure 56, the values turn are calculated in figure 57. Three values turn into 1 and will therefore get a force on its vertical DoFs.

To calculate the force for F$_e$, the maximum snowload for each element is divided by 4 and then placed on the DoFs. For the element the force is:

$$F_{nodesof[i,j]} = \frac{snowFactor}{4} f(y_{[i,j]}) \quad [35]$$

For an element the total force that is playing can be noted as follows:

$$F_e = F_{preset} + F_{self} + F_{snow} \quad [36]$$

Where F$_{preset}$ is both the preset point and roof loads. In order to calculate the compliance, the total force, F$_e$, has to be taken. This is easy to compute as it is the sum of all the other forces in equation [36]. It is important to look at the sensitivities, as this is less easy.

F$_{preset}$ is not dependent on x and therefore the derivative of F$_{preset}$ is 0. The derivative of F$_{selfweight}$ has been calculated in the previous toy problem. The derivative of F$_{snowload}$ has to be calculated as well. The derivative of the snowload can be written as:

$$\frac{\delta F_{e[i,j]}}{\delta x_{[i,j]}} = \frac{snowFactor}{4} \frac{\delta f}{\delta y_{[i,j]}} \frac{\delta y_{[i,j]}}{\delta x_{[i,k]}} \quad [37]$$

$$\frac{\delta y_{[i,j]}}{\delta x_{[i,k]}} \begin{cases} 1 & k \geq j \\ 0 & k < j \end{cases} \quad [38]$$

Or in other words; the value of dydx is 1 when the sum of elements above the element (excluding the current element) is smaller than the element itself. Using this, only values that are above the roof are given a sensitivity and generated geometry cannot . The second piece of the derivation is:

$$\frac{\delta f}{\delta y_{[i,k]}} = \frac{\delta HS}{\delta y}(y - 0.5) - \frac{\delta HS}{\delta y}(y - 1.5) \quad [39]$$

$$\frac{\delta f}{\delta y_{[i,k]}} = \frac{100}{(\pi * (10000 * ((y_{[i,k]} - 0.5)^2) + 1))} - \frac{100}{(\pi * (10000 * ((y_{[i,k]} - 1.5)^2) + 1))} \quad [40]$$

This is implemented in the system, together with the roofconstraint and the selfweight. It is expected that the geometry is a bit thicker, as result of the selfweight problem was to add less weight and follow the catenary curve. Figure 59 shows the final result that was generated including the snowload. The result looks very similar in shape as compared with figure 55, but there is an obvious increase in volume. The arch still looks a lot like a catenary curve, but is flattened out a bit. The final compliance is a bit higher than expected, which is due to scattered material in the void. The following page will contain some analysis on the results.

First the result of the selfweight is exported to Rhino and an average surface is made by using the control point curve. This surface is then exported to Ansys and placed in the XY plane. In the model is meshed using linear meshing and fixed supports are added. Lastly the global gravity is added and because Ansys needs at least 1 force, a force is place with a very small value on the top of the arch. The calculated stresses can be seen in figure 61, which are actually quite a few. Especially halfway the arch, quite large stresses arise, which leads to some deformation of the arch. This is explainable by the fact that the algorithm will try and find the absolute lowest compliance, which is found at the lowest force. So minimizing the material in a way that almost no forces exist anymore, is a valid solution, but might not be realistic. To make the arch a bit more realistic, the snowload is added.

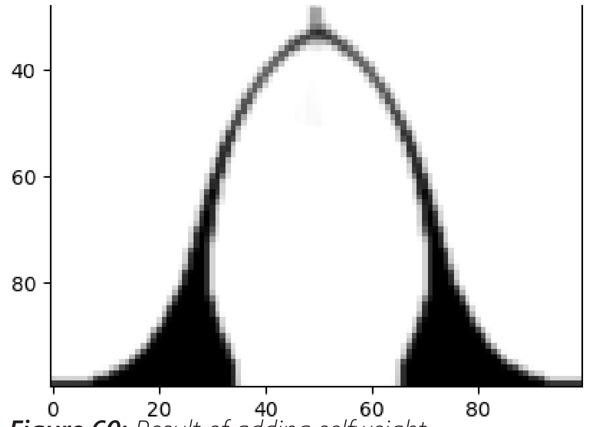


Figure 60: *Result of adding selfweight*

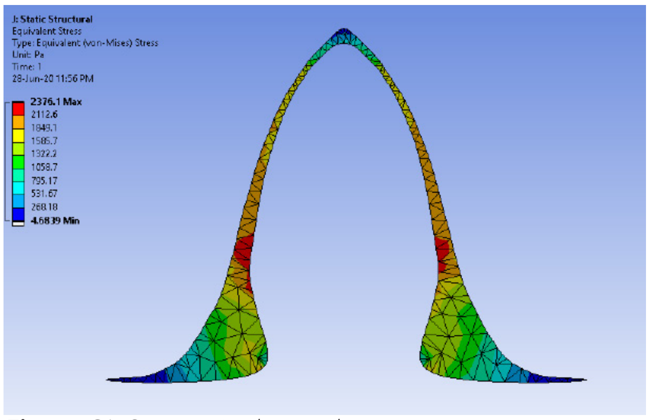


Figure 61: *Stresses in the result*

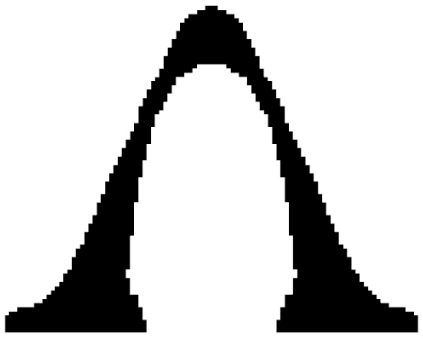


Figure 62: *Shape under selfweight and extra forces*

The result of adding the snowload results in more material and a thicker construction. As there are now extra loads that can play on the system and this requires more material. Having to carry a load with little material will result in a higher displacement. As the displacement is directly influencing the compliance, this is very bad for the system. The result shown in figure 62 only adds around 1/16th of the total selfweight, which seems realistic to real scenarios.

It has to be noted that figure 62 has been slightly modified, as there was a lot of scattered material in the rest of the design space. When all the elements with an density higher than 0.8 were chosen, figure 62 was the result. Also, the snowload seems to behave very weird with certain values of `snowFactor` and often resulting in a very bad solution.

>>>3.5.5 The bus station

Now that the possibilities of the 2D algorithm is tested and developed, the algorithm is translated into 3D. Chapter 3.2.3 already summarized the changes that Liu and Tovar made in their paper, in order to write a Matlab code. This chapter will further elaborate on these changes and the next will discuss the implementation of selfweight and snowload as well. Working with this algorithm is more tricky, as test could take very long. During the development also some optimization was performed, of which the details can be found in chapter 3.7.

The first step in order to retrieve useful results is to translate the basic script, that was used in TOY-problem 1, to 3D. Liu and Tovar (2014) wrote a translation of this code in MATLAB and together with the developed code, based on Deetman (Deetman, 2019). Overall can be said that the algorithm functions the same way and 2D arrays are made 3D. The main difference between the two translations is the indexing system, which is key to solving 3D problems.

The basis of this indexing system is the way that `numpy.reshape` handles the translation from a vector to a 3D array. Following this indexing method is the most convenient and fastest way to translate the many vectors that are used in FEA, to the 3D field. Figure XX shows how the translation from a vector to an array is made and the result of the numbering. Chapter 3.2.3 discussed how Liu and Tovar solved most of the indexing issues, and how the translation to Python was made.

Configuring the inputs correctly is essential into getting good results and later on further developing the algorihm. The loads and supports will be configured the same way as in 2D topology optimization, where a force-vector is created and fixedDoF-vector. The force vector will contain values of 1 at the DoF where force is placed. A force can have all the directions, but usually only vertical forces are used. The same goes for the loads, which are configured as in 2D. The formula for calculating the corrosponding DoF is done as follows:

```
nodeID = z*(nelx + 1)(nely + 1)+ x(nely + 1)+ nely-y
```

The DoFs can be retrieved and follow the following table.

Direction	DoF-ID
X	3 * nodeID+1
Y	3 * nodeID
Z	3 * nodeID-1

Using these vectors, the first algorithm can be tested with some values comparing with the results that Liu and Tovar created. Figure 64 shows a simple beam with a single force and the 4 lower corners as supports. It looks to be a very stiff and is very believable to be an optimized beam.

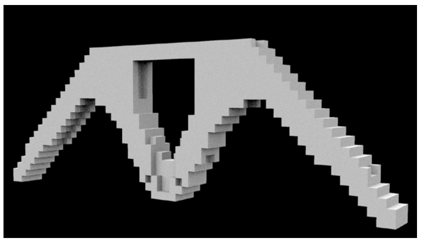


Figure 64: *3D beam using TO*

The next step is to implement the voids and area loads, as was also performed in TOY problem 2. After the system is optimized by MMA, the element density is a vector with the length of nele. Because of this property, a list of voids can be created (in Rhino) and imported in the algorithm. In Rhino it is important to follow the same indexing order to check if a point is inside the chosen void. The following pseudocode shows the proper order:

```
voidList = [ ]
for each element in the designspace:
    isPointInside = void.contains(element)
    nodeID = yvalue*((nelx)*(nely)) + xvalue*(nelz)
                         + (-nelz + nely)-1
    if isPointInside == True:
        voidList.addtoList(nodeID)
```

Pseudocode 20: *Creating list of voidIDs*

The output of this algorithm is a list of nodes that will be implemented with the following lines in pseudocode 21. This is identical to the 2D solution for voids.

```
voids[voidslist]=1
where voids = 1, x = 0.001
else: x = x
```

Pseudocode 21: *Setting voids to the minimum values*

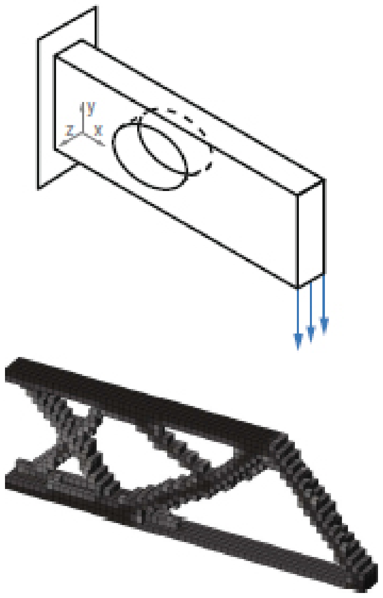


Figure 65: *Voids in a cantilevered beam*

The following pictures show some 3D examples of geometry, using voids inside the designspace. Figure 65 shows a cantilevered beam with a round void in the middle of the beam. Figure 66 shows some exploration that was done with a small house, to figure out how doors and windows could work.

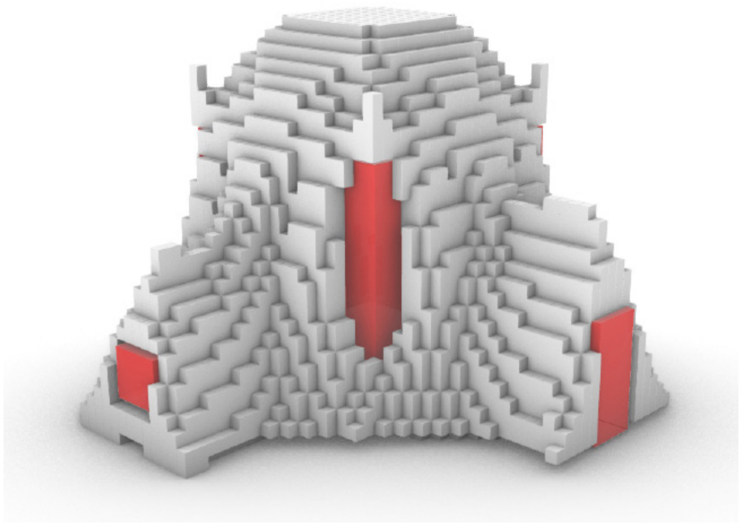


Figure 66: *Example of voids being used as doors and windows*

In order to further develop architectural geometry, area loads have to be added. Implementing area loads follow the same rules as in 2D and the same procedure. Of all the void groups, the values with the lowest Y-coordinates will be selected and their bottom nodes retrieved. A force is placed on all the vertical DoFs. In order to check if the area load is applied correctly, a roofconstraint is applied on a design space with a size of 3:1:10 (y,x,z). The supports are placed somewhat like the QNCC in Qatar, which is the largest building created with Topology Optimization (Naboni, 2018).

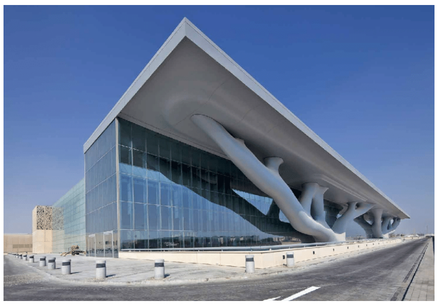


Figure 67: *QNCC, worlds largest TO building*

It can be seen that the roof correctly takes the area load, as it is distributed over the full roof. This leads to a thick slab under the load, which is comparable with figure 67. Two supports were set and from there two arms arise. Both are symmetric, which makes sense as the whole system is symmetric. As the QNCC has many more variables and another way of calculating, the shape is different. But overall the shape is very simililar.

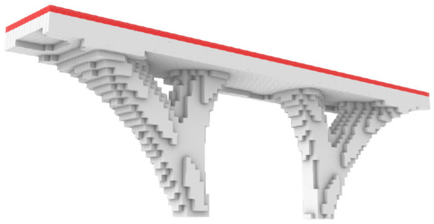


Figure 68: *Comparable design using this research*

Lastly, all previously described additions of the algorithms are combined in order to design a small busstation with it. The supports are set to the full XY plane and void is added on the side of the design space. Above the void, at Y = nely, an area force is placed that will act downwards on the void. This force can be seen as a temporary roof constraint. Figure 69 shows the final result from the front side, where figure 70 shows a section. The force is distributed very interestingly, which the section shows clearly. One array of columns takes the force to the outer edge of the design space, while the other goes inwards. Figure 69 shows that behind the void, this is also shown. Also above the void, no material is placed directly, which saves a lot of material. Lastly is it interesting that the front is generated, but only as a sort of fake facade. Most likely forces close to the front edges will be transferred in this piece of geometry

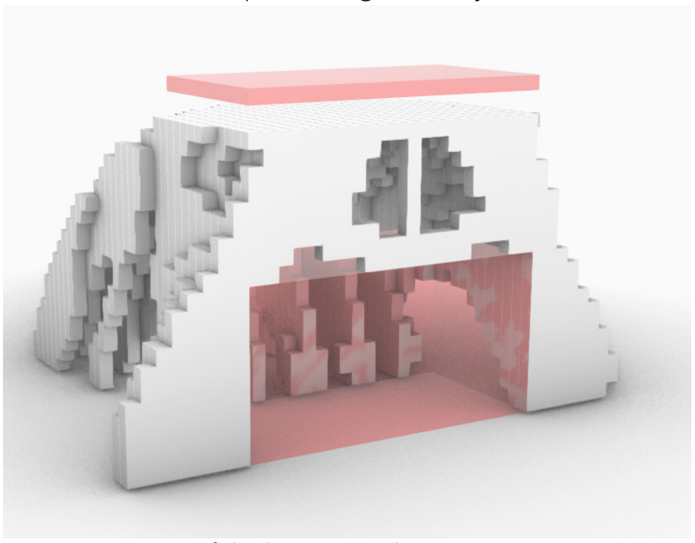


Figure 69: *Front of the busstation design*

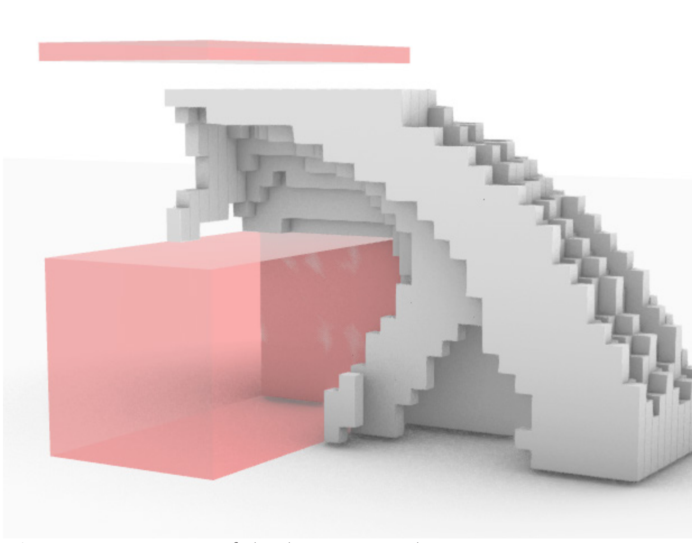


Figure 70: *Section of the busstation design*

>>>3.5.6 A small house

In this last TOY problem selfweight, the roofconstraint and the snowload are introduced to the 3D algorithm. All of them will follow the same theories that are described in toy problem 3 and 4, this chapter will discuss the implementation and its outcomes and limitations.

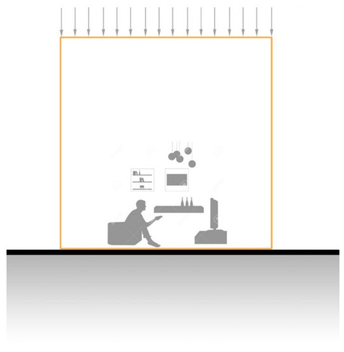


Figure 71: *TOY Problem 6*

First the selfweight is introduced, following the same formula as in chapter 3.5.3, equation [21]. Note that the force is divided over 8 elements in stead of 4. The density is, just as in 2D, a 1D vector and adding selfweight isn't that hard because of this reason. Finding the derivative of the selfweight also follows chapter 3.5.3, as the derivative of the compliance is:

$$\frac{C_e}{x_e} = -px_e^{p-1} - U_e^T \frac{\delta Ke}{\delta x_e} U_e - 2U_e^T \frac{\delta Fe}{\delta x_e}$$

Where dFdx is a vector with the selfweight on its vertical DoFs. The only change that has to be made is to create a vector with every third value of the selfweight:

```
selfweightvector[1::3] = swfactor/volfrac*nele/8
```

Figure 72 shows toy problem 5 with selfweight, which shows a very comparable geometry as in the toy problem. The main difference in the front is that a lot of unnecessary densities are removed an the whole building has become a lot more round and organic, which is shown in figure 73.

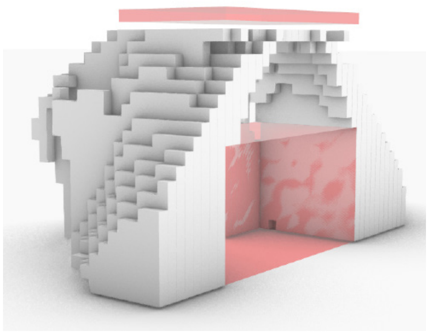


Figure 72: *The busstation under selfweight*

Secondly the roofconstraint is implemented, in order to not need preset forces anymore. Due to the numbering and indexing of 3D, the method that was created in TOY problem 4, isn't usable anymore. Mathemetically, the constraint is the same as in 2D:

$$KS(x) = \frac{1}{P} ln(\sum_{k \in K} e^{Pg_k}) \leq 0 \qquad [27]$$

Due to the numbering of the 3D space, x can only be reshaped into a nely,nelx,nelz array, in order to keep all the values at the right space. Where previously a list could be made of the columns, now this list is an array. Due to this being much harder to implement, another approach is taken. A new matrix is created with the shape nely,nelx,nelz, which is called aboveVoid. This value contains values of 1 directly above the void, and 0's elsewhere. Using this matrix values outside this area can always be set to 0. Pseudocode 22, on the next page, shows how it is further performed.

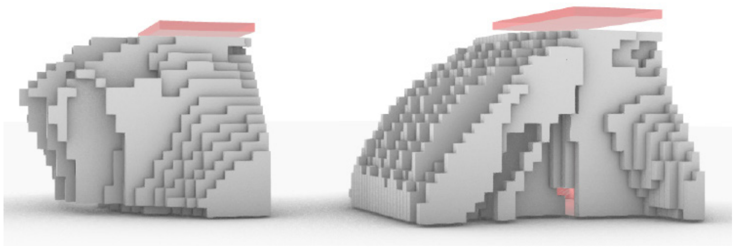


Figure 73: *The busstation under selfweight*

```
invertedVoids = 1 - voids
voidColumns = tile(bottomOf(voids), shape=nely,nelx,
                                               nelz)
onesAboveVoids = invertedVoids * voidColumns
#Calculate constraint and sensitivities
epgk = e ^ (10 * (1 - x ^ p))
gcolumns = log ( sumAll(epgk)) / 10

#Construct final matrices
repeatDc = repeat(epgk / gcolumns, amount = nely)
dcrc = repeatDc * onesAboveVoids
```
Pseudocode 22: *3D roofconstraint and its sensitivities*

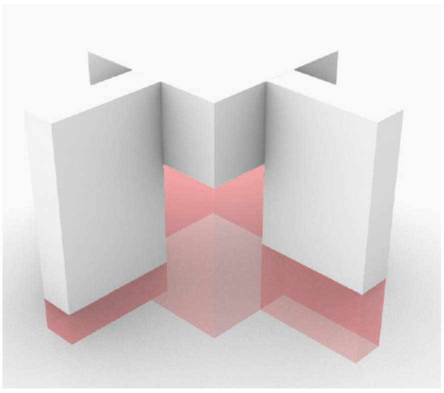


Figure 74: *Above the voids the roofconstraint is set*

Implementing the constraint decreases the speed of the algorithm significantly. Therefore, the constraint factor has been lowered to 0.001, as it seems to still provide believable results. Figure 74 shows a design space where the constraint is in function.

Combining selfweight and the roofconstraint will lead to a design, whenever at least a void is placed. Assuming that the supports are the vertical DoFs in the entire XY plane, the roofconstraint will fill up the area above the voids. The next iteration, this filled up area will gain selfweight and this is processed to the supports. This should result in an, as much as possible, compression only building.

Lastly the snowload is added, which causes more calculations to be made and an even slower algorithm. Due to the fact that a high resolution is preferred over the existance of snowload, further calculations don't include the snowload. Implementing the snowload will follow the identical steps as in 2D, where it is important to take the sum of the elements over axis = 0.

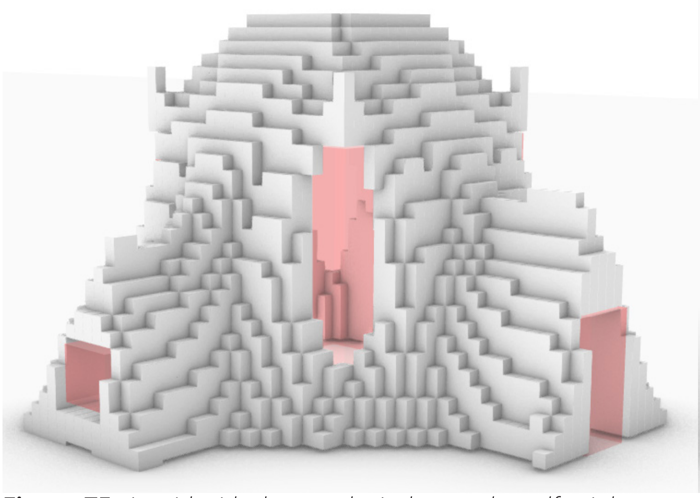


Figure 75: *A void with door and window under selfweight*

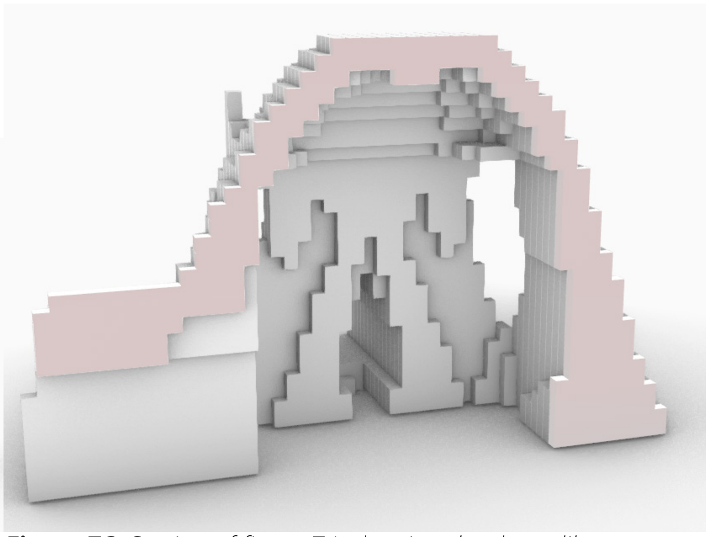


Figure 76: *Section of figure 74, showing the dome liks structure*

>>>3.6 Architectural implementations

In this chapter the 3D algorithm is taken and several inputs are explored. The inputs, mainly consisting of voids and forces, are configured in a way that represent architectural cases. Voids are considered as living spaces, doors and windows. This chapter will provide insights in how these shapes are generated and what is to learn from them

A small house:

The previous page already showed the house that was generated using a void with two extruded voids, functioning as a window and door. The final result can be seen in figure 75 and 76. Figure 75 shows the overall shape of the building with the colored voids. As the middle void was quite small in comparison with the design space, the window and floor are quite long. The door has become a small hallway, of which figure 77 is a small detail. The outside shape shows and arch shape but probably due to the square shape of the void, the inside looks quite sharply cut off. The inner section of figure 76 shows that around the entry to the window, and extra arch seem to be placed. This shape seem to have an arch on the inside, but the resolution of the voxels is too low to conclude from this. Also the section shows that the hallway is quite shaply cutoff, but in the beginning some arch is formed. The results shows that cubic voids are not ideal, as also the edges of the voids show by cutting through the dome. A wall constraint could be an option, but will mostly decrease the strength of the building. Masonry building doesn't go well together with cubic shapes.

The section in figure 76 does also show the overall section of the building, which represents the shape of a dome. As domes are usually constructed for compression only masonry buildings, this is a quite interesting result, which shows that the algorithm gives quite reliable outcomes.

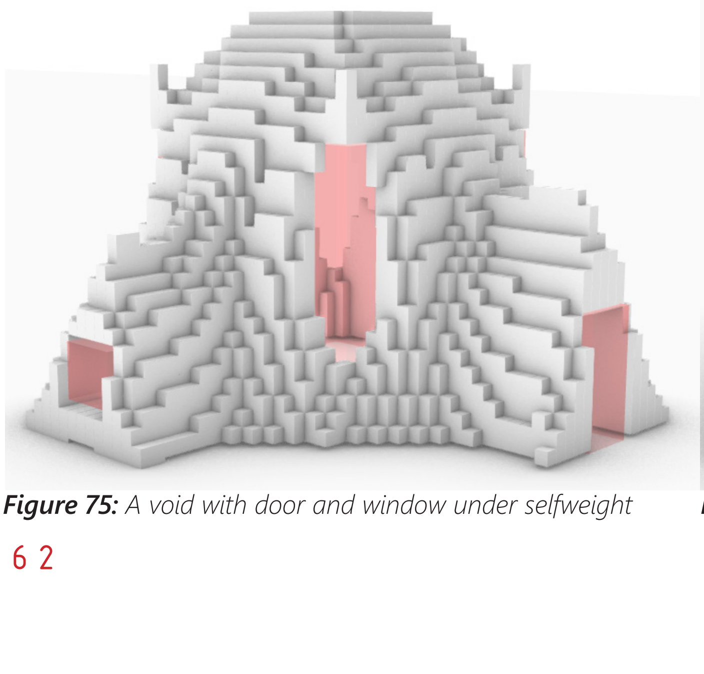


Figure 75: *A void with door and window under selfweight*

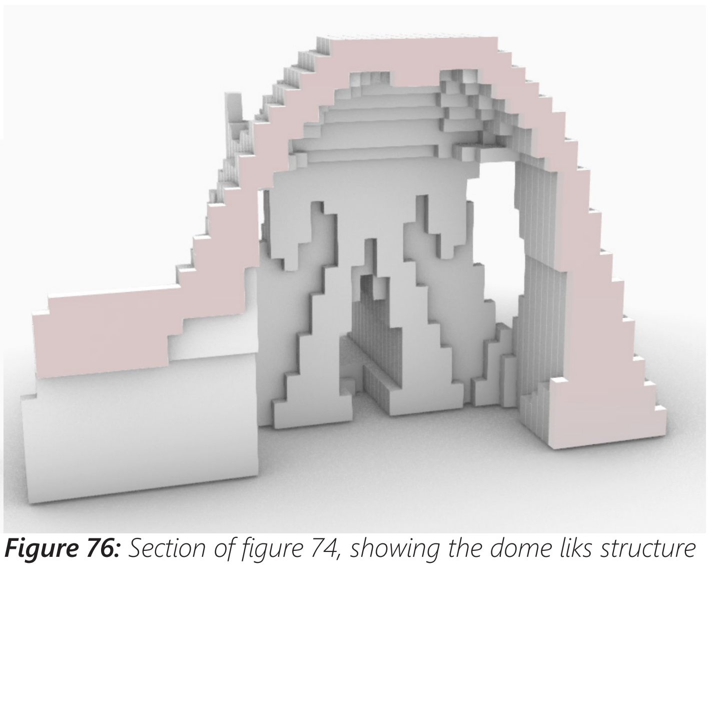


Figure 76: *Section of figure 74, showing the dome liks structure*

Domes and arches are very interesting architectural elements as they are used for so long and still today many buildings are built using them. In compression-only buildings, they are the key element in making roofs and often are really beautiful.

To futher research the creation of domes in topology optimization, a dome is generated with a higher resolution, namely 50x50x30. A higher resolution will better show how generated domes could look like and how they could be the output of this algorithm.

Figure 78 shows the section of the dome which looks very convincing the shape of a masonry dome. The critical point is where the squared void touches the dome and cuts a bit in the domes shell. The thickness of the structure is only 1 voxel at this point which is probably a very poor structural design. The geometry seems to counter this by having a mainly vertical direction in this area and the beam staying on roughly the same column.

The geometry can be seen as two parts, the columns on all of the sides and the dome on top of the void. Due to the shape of the square, the columns are parrallel to the void, which is unusual for domes. Usually on the corners these columns are placed and the areas parrallel to the shape are offset much more than in this case.

The dome above the void is octogonal and looks in section very much alike regular domes. The gradient is very similar to known architectural examples. The geometry follows rules of masonry building, but with the limitation of rectangular voids, domes aren't generated in an optimal shape.

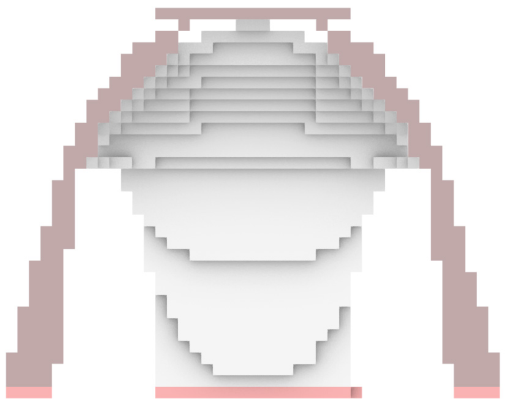


Figure 77: *Section of the larger generated dome*

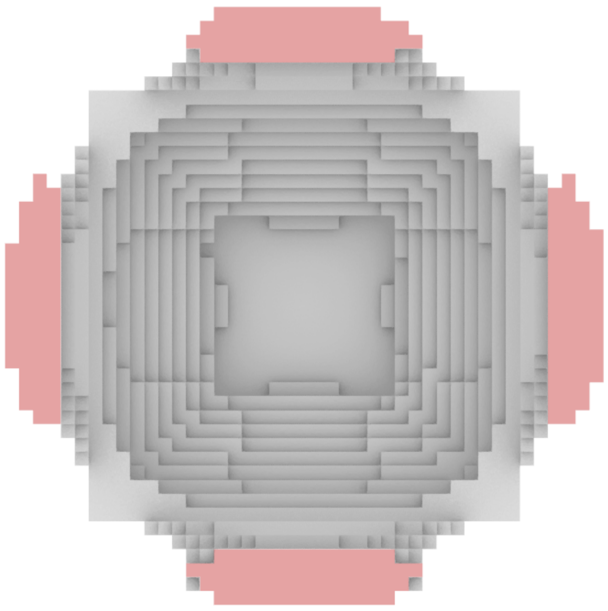


Figure 78: *Octogonal pattern in the big dome*

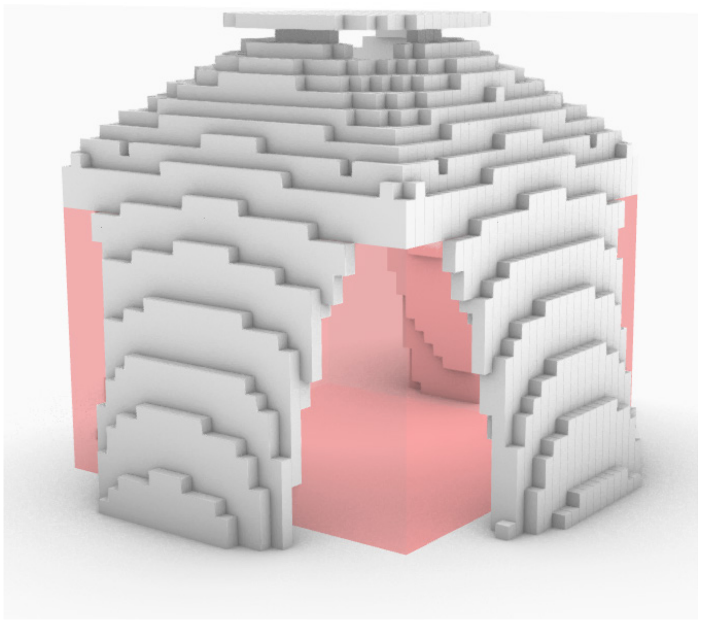


Figure 79: *Perspective of the dome, showing the columns parrallel to the void*

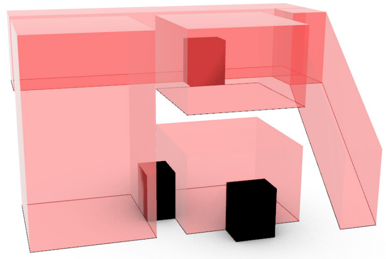


Figure 80: *Configuration of a 2 story complex house*

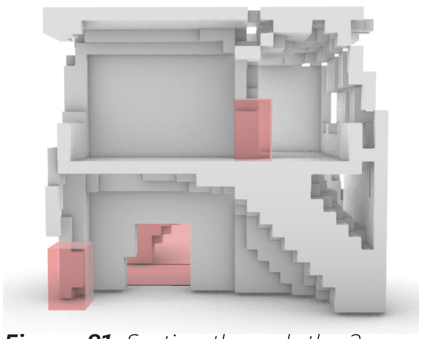


Figure 81: *Section through the 2 stacked voids*

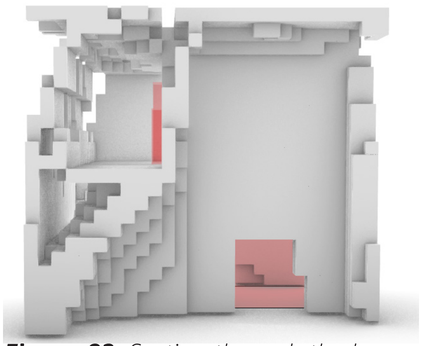


Figure 82: *Section through the large void*

A 2 story complex house
To explore the extents of the algorithm, a complex house is analysed. The design, shown in figure 80, is developed by Ivan Avdic (2019) and gives several difficult to solve problems in one design. The design space exist of a rectangle that represent a two story house. The bottom floor has two rooms, of which one is extracted to the roof. Above the other void is another story, which is accessed by a gallery with a stairs.

Challenges in this design are the proper configuration of the voids, the forces that raised voids have, the difference between stacked voids and extruded voids and the implementation of the doors. Overall, it is an complex building that when solved, should prove that the algorithm works sufficiently.

Using an area load on the roof will give the result as shown in figure 83. All the voids are configured correctly and the main force, the area load, seems to be transferred correctly to the ground. The sides of the design, as seen in both perspectives, look optimized for the force. In the outer walls holes appear where no material is needed and in the large void no facade is placed. This makes sense, as it is more optimal to combine the load from the roof and try place it on two walls in stead of 4.

Sections show that voids are properly placed and the hallway is cut out properly. Doors inbetween voids are also left open and even are higher than needed. Below the hallway the design space is empty, but this is optimized by culling out material. For the voids on ground level, this cull is half a dome, which makes sense as selfweight is implemented in the design.

The main problem in this design is the area load, as it is place just right above the voids, resulting in a flat roof. This is not optimal, so this area load is replaced with a roofconstraint.

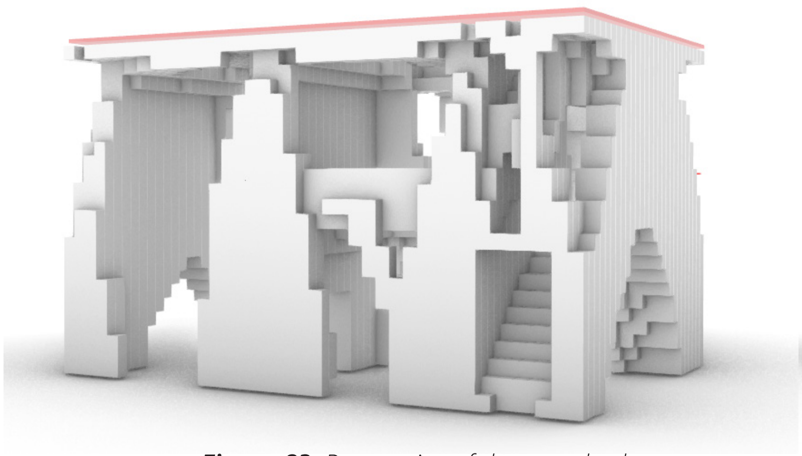


Figure 83: *Perspective of the complex house*

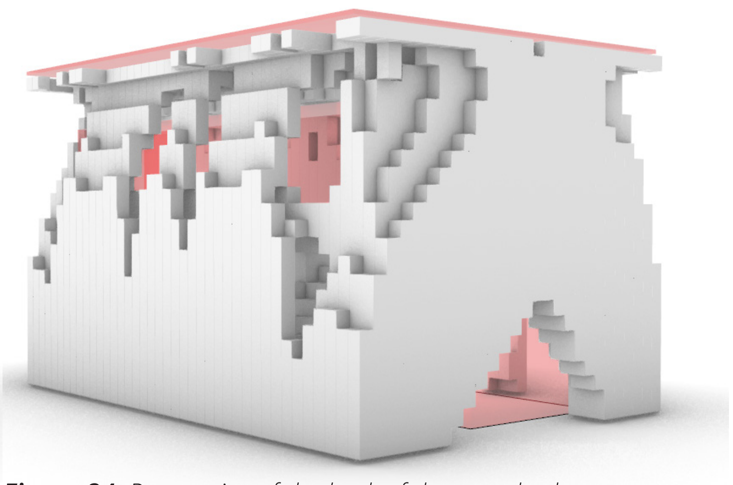


Figure 84: *Perspective of the back of the complex house*

Implementing this roofconstraint follows the steps explained in toy problem 6. Figure 87 shows the perspective of the resulting geometry, which seems to follow the voids properly. The stair is generated according to the void, shown in figure 86, and also the hallway is placed correctly. The force above the hallways seems to have been incorrectly, as a flat roof is built and this roof is pushed through the space above the void. This inaccuracy can also be seen below the hallway which seems to not be optimized. This is most likely caused by how the roofconstraint is calculated and that raised voids arent calculated properly. Nevertheless all the voids have a roof over its elements so the roofconstraint is fullfilled.

When looked at figure 86 and 87 all the voids have a roof above them and they are shaped as small domes. This is to be expected, as this is mean to happen. One constraint that has not been implemented is a roof constraint, which is clearly shown around the larger void. Not placing any walls is good for the volume and the forces seem to be transferred through the sides of the voids. A wall constraint could be implemented, but this shape is technically sufficient. Therefore, a wall can be placed, but doesn't need to be load bearing. Material wise, it could be even a large window.

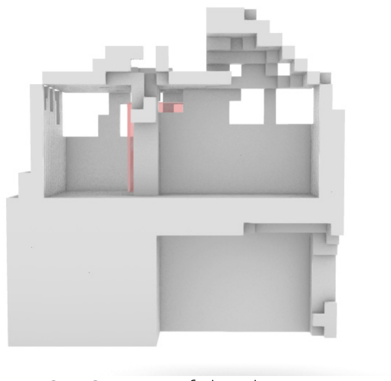


Figure 85: *Section of the design*

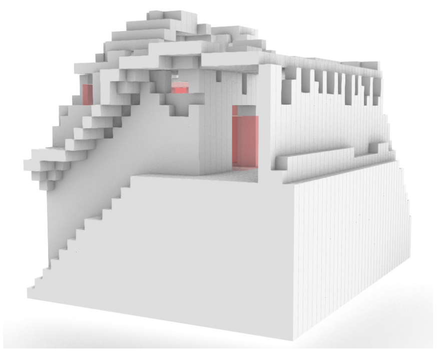


Figure 86: *Section of the stair and hallway*

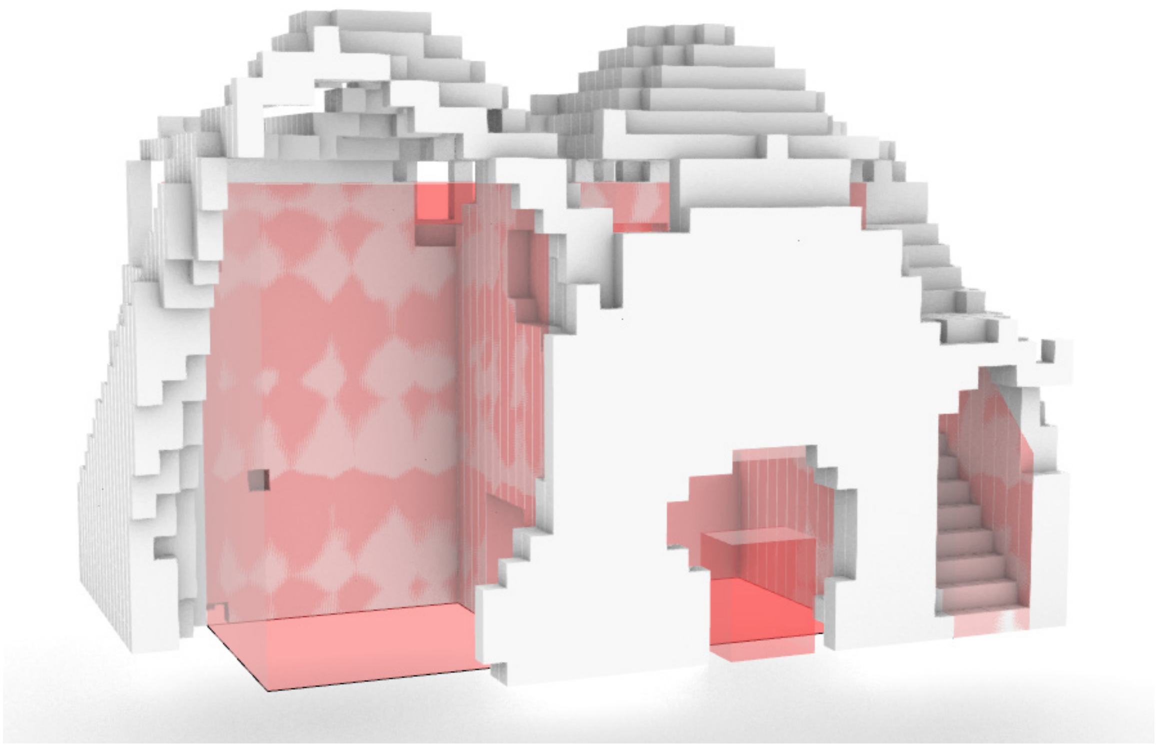


Figure 87: *Perspective of the complex house with roofconstraint*

Figure 88: *Haus am Horn*

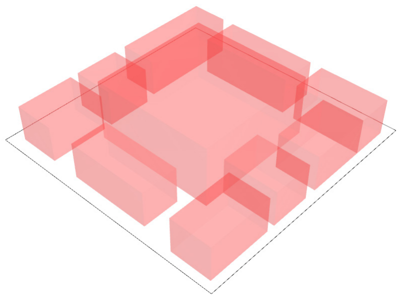


Figure 89: *Void in the design space*

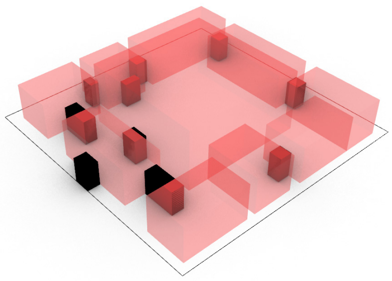


Figure 90: *Doors in the design space*

Haus am Horn
To see how the algorithm compares with architecture and to answer the question *"How can topology optimization transform the design process of masonry buildings, particularly their configuring and shaping processes?"*, an architectural example is analysed.

The Haus am Horn, designed by Georg Muche, is a modern villa that is an example of Bauhaus. The building exists of one main living area which is raised an higher than the other rooms. Around the living area some smaller areas are placed with doors inbetween them. Figures 89 and 90 show the voids and the doors in the design space. In the design space no forces are placed, only the roofconstraint is placed on the voids, which should result in a final shape.

Figure 91 shows the final calculation, which shows all the walls being filled with elements and smaller domes on top of the rooms. The rooms being filled is also shown in figure 92, which shows a plan view of the building. Doors are implemented and show small arches above them.

Looking at the sections again it can be seen that domes are created. The larger and middle room is generated as a large dome and the much smaller rooms are arches (in figure 93) and in the corners they are domes (in figure 94). Overall, the modern look has been replaced by domes and a much larger overall shape. The layout of the design is intact. The overall shape could be improved by changing the voids to non cubic shapes.

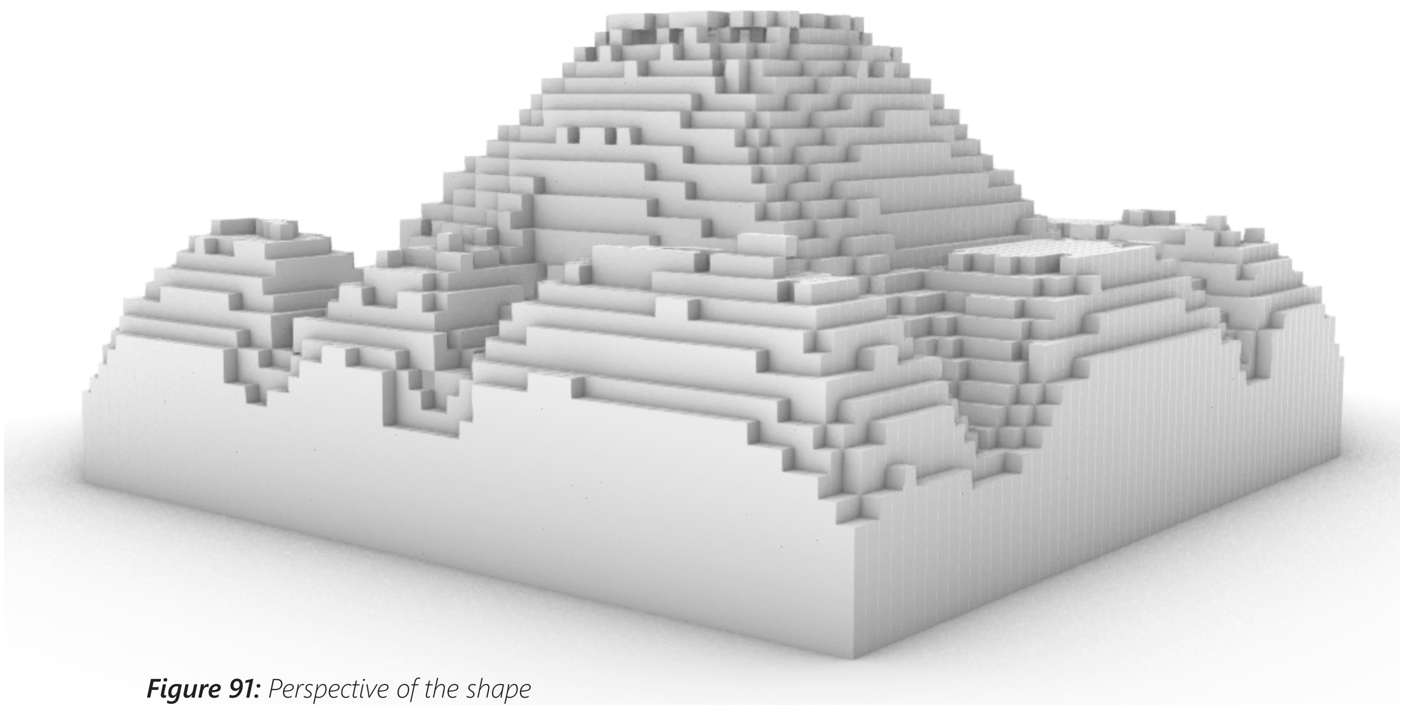


Figure 91: *Perspective of the shape*

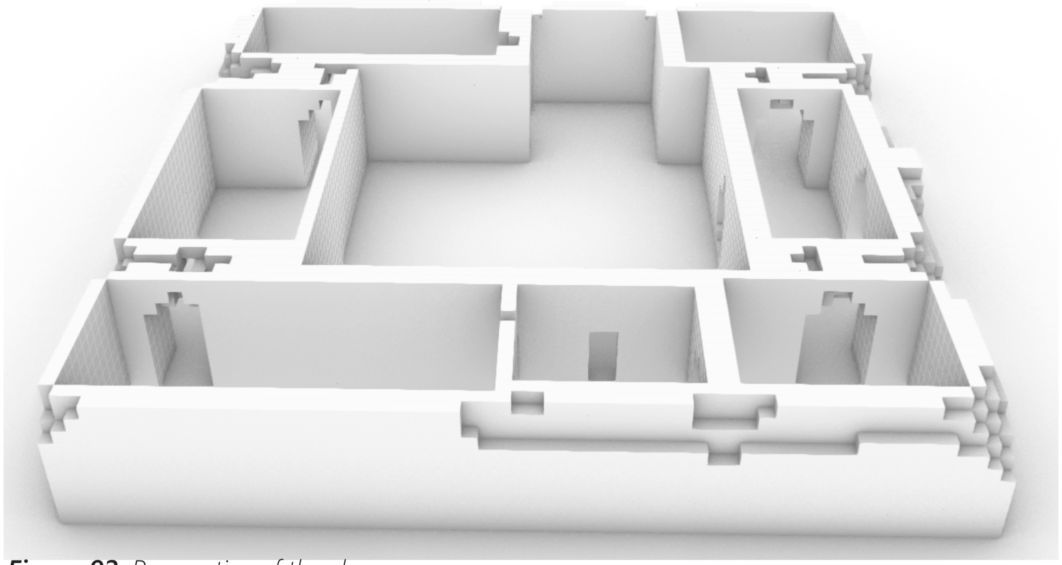


Figure 92: *Perspective of the shape*

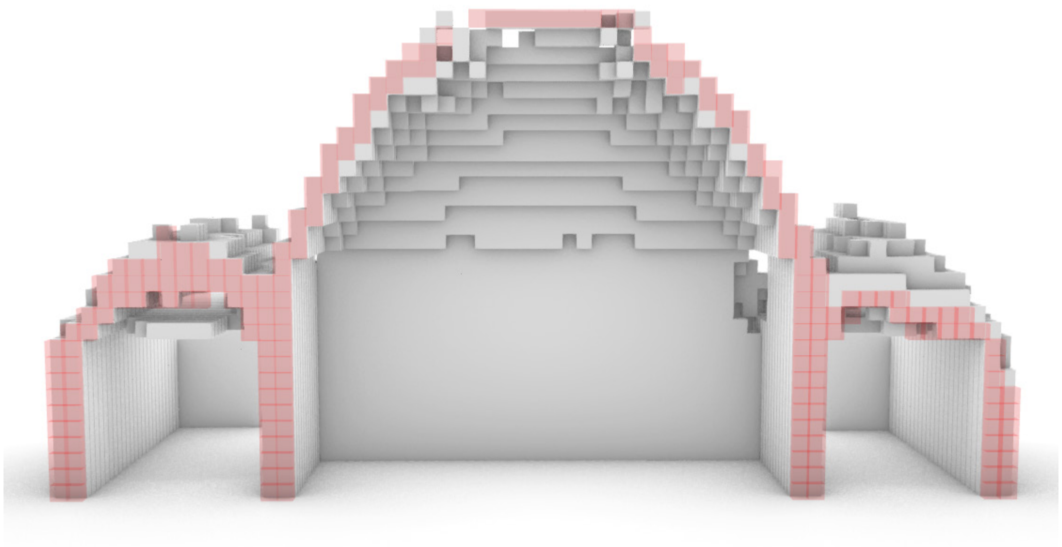


Figure 93: *Section through the middle void*

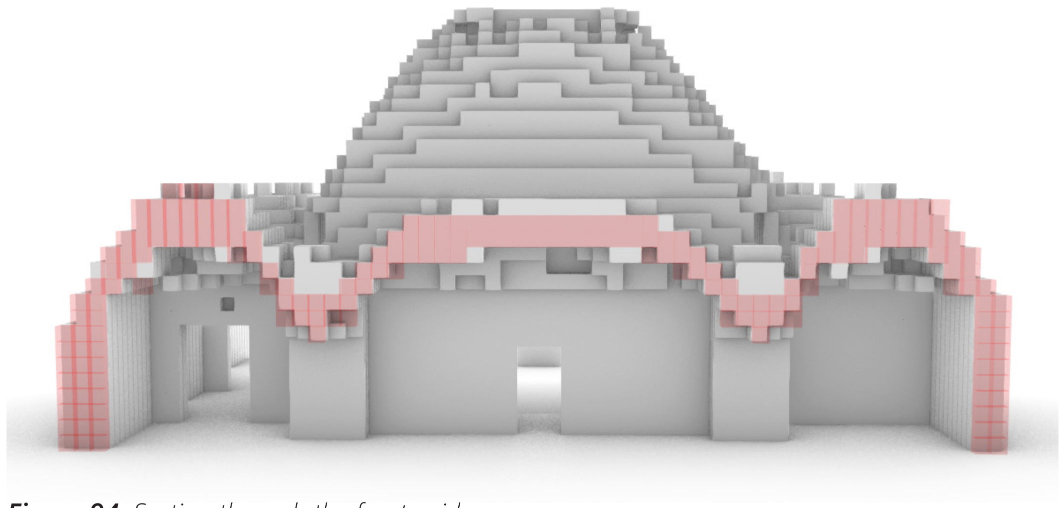


Figure 94: *Section through the front voids*

>>>3.8 Code optimization

Optimizing the code is essential in order to get this algorithm to work in Grasshopper. Grasshopper is more sensitive for bad written code, so increasing computational time is key in order to succeed. The 2D code works quite good, as the computational time for quite big design spaces are quite good. The 2D code is very efficient, mainly using matrix multiplications and the smart use of sparse matrices. Mainly the 3D algorithm is very slow, which asks for optimization. This chapter handles this optimization and how this could work more efficient.

In order to check the time each line takes, a profiler is used. Python provides a C module called cProfile, which is imported (Danjou, 2015). cProfile allows for an entire script to be analysed and outputs the cumulative time for each function. To test the speed of the algorithm, a large system is chosen with 500 x 500 elements and the profiler is run.

In the beginning most changes are made using simple for-loops which, especially in 3D, are very inefficient. Initially, selfweight was calculated in a for loop where the value of x is checked to be some threshold. Especially in large systems, this takes long amounts of time. Implementing a heaviside smooth function, allows this system to quickly update the selfweight. Another improvement is made by calculating the preset F vector before the first iteration and saving the value in a new vector F_{preset}. Each iteration this vector is called and added to the calculated F vector. Previously the F_{preset} was calculated each iteration, which takes unnecessary amount of time.

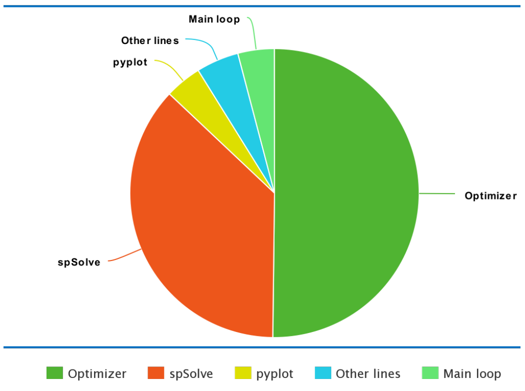


Figure 95 *PIe chart of calculation times*

After making these changes, the cumulative times of the algorithm are shown in the graph in figure 95. The largest portion of the time is spent by numpy.sparse.linalg.spsolve, which is the solver to solve K=F/U. Decreasing the time that this function takes, will greatly reduce the time the total algorithm takes. The first improvement that could be made is an more efficient numbering method, such as the minimum degree method. However, this is by default chosen in spsolve, so there will be no gain using this (Scipy.org, 2020).

Looking at the solver, SciPy offers a few other sparse solvers that could be used. Most notably are the generalized minimal residual solver, or gmres, and the conjugate gradient solver, cg. Gmres

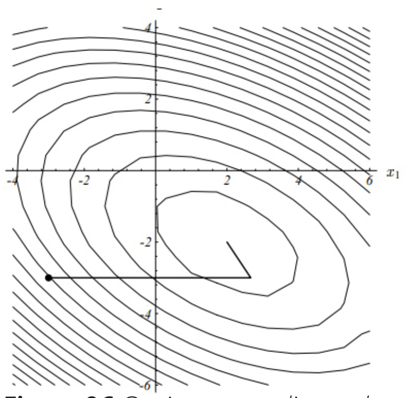


Figure 96 *Conjugate gradient solver*

The conjugate gradient solver works quite alike the gradient descent optimization method. In gradient descent the slope is calculated at x_k and a step is set into opposite direction of the slope. Eventually the optimal solution can be found (Michailidis & Maiden, 2013). In the conjugated gradient method the steps are chosen, such that the following step will lead to the minimum. A step is taken along the gradient, where the gradient at the following value is orthogonal and leads to the minimum value (Shewchuck, 1994).

In the lecture notes by Sleijpen and van Gijzen (2017) a flowchart is presented on what solver to use to solve Ax = b. As matrix K is both symmetric and larger than 0, it is best to use the conjugate gradient solver.

To implement cg in the algorithm the following paramaters have to be configured properly in order to get the quickest results:

scipy.sparse.linalg.cg(A, b, x0=None, tol=1e-05, maxiter=None, M=None, callback=None, atol=None)

Configuring these parameters will greatly increase the efficiency. There are 3 main parameters that are important, x0, tol and M. x0 is an initial guess towards the result, which makes it easier to find a result. Tol is the tolerance for convergence, where a lower value will accept results quicker. Lastly M is the preconditioner.

A preconditioner is a matrix that is chosen such that:

$$M^{-1}Ax = M^{-1}b \qquad [41]$$

The preconditioner is a matrix that is easy to construct and will greatly increase the computation time. Many preconditioners exist, where this research will focus on two of them. Jacobi preconditioning is a very quick to compute matrix that follows (Sleijpen & van Gijzen, 2017):

$$M = diag(A) \qquad [42]$$

Secondly ILU-preconditioners can be used, which are the most popular 'black box' preconditioners. Using this preconditioner, the matrix M will look like (Sleijpen & van Gijzen, 2017):

$$A = LU \qquad [43]$$

This preconditioner is implemented in scipy.spilu and has its own solver, which will be used in the test. For the cg method, the Jacobi preconditioner is constructed, using equation [42]. Some testing with the solvers showed that tol = 1e-03 will still give results which are accurate enough. As the main algorithm has many iterations, these inaccuracies will flatten out. Lasly x0 is chosen to be u_{k-1}, the deformations in the previous iteration. All the possible solvers are tested for 20 iterations in a 500x500 system. The results are shown on the next page.

spsolve(K,f)	gmres(K,f)	cg(K,f,M=1/Jac)	cg(K,f,M=Jac)
27.58	511.49	36.57	34.96
38.52	0.07	0.06	0.05
32.44	0.06	0.05	0.05
33.81	19.14	8.04	8.58
25.90	0.06	0.06	0.06
30.91	19.76	9.94	11.18
28.68	0.06	0.06	0.06
24.20	22.70	11.10	12.97
26.82	0.10	0.06	0.06
30.58	26.83	12.04	15.37
26.84	17.90	9.33	0.05
26.19	19.13	10.37	18.31
26.82	20.51	10.66	16.78
24.33	20.48	10.94	18.04
23.92	19.94	11.21	20.37
23.66	22.06	11.39	22.43
21.74	20.08	11.50	25.07
25.66	20.17	11.73	28.59
24.44	22.22	11.88	31.37
24.42	22.51	12.44	37.72
Total(sec)			
547.44	805.96	189.43	302.07

Figure 97: Time spent calculating per solver

Looking at the values that are calculated in figure XX, it can be clearly stated that the conjugate gradient solver works the most efficient. All the solvers have a slower first iteration, as in this iteration there is no x0. Especially gmres is extremely slow here, where perhaps another preconditioner should lower this time.

The difference between the conjugate gradient solver with the right preconditioner is also very clear to see. In the early iterations not much difference is seen, but once large changes in the system are gone, the right preconditioner seems to be the Jacobi preconditioner. Usually solving the system takes much longer, which results in this solver being even more efficient.

Note that these timings are in seconds and a system of 502.000 DoFs is solved. For 2D optimization this is a very high resolution, and these times are similar to a cubic design space with sizes of 55. A system like this is expected to take another 80 seconds to solve, resulting in 5 minutes of calculation times, purely for solving the system. Adding the MMA solver in this equation will increase the calculation time to roughly 20 minutes.

>>>3.9 *Plugin development*

The goal of this research was to implement topology optimization in architecture, or at least try to set some steps into this direction. Topology optimization might be a great way to generate shapes and should also be available to architects and architecture students. To achieve this availability, a plugin is written that could be installed in Grasshopper. Grasshopper and Rhino are programs that are used quite often in architectural practice and are taught at universities. Pursuing this goal would allow students to see the possibilities of topology optimization and generative design. This thesis is meant as a guideline in order to show what happens inside the black box.

In order to write the plugin, first a flowchart is developed in order to note all the functionalities that the plugin should have. Plugins like Karamba3D are comparable to how this plugin should be used and these plugins are looked at when developing this plugin. One key feature that Karamba3D has is a model viewer, in order to check if the model is properly composed before starting calculations. Chapter 3.6 showed that computational time is the hardest factor of this code and the main obstruction for complex problems. Grasshopper has the tendency to crash (actually *waiting to respond)* while performing calculations, until the calculation is finished. Because of the computational time and the behavior of Grasshopper, such a model viewer is important to have.

The flowchart on figure XX shows the steps of assembling this model and how the viewer should be updated. After the input of each element, the element should be added in the viewer and / or possible errors should be showed. Properly assembling the model is the most important thing and should be intuitive, When all the inputs are assembled and no errors arise, the model can be inported in the solver. There should be a lock that the solver will not work, unless no errors, *errorcount = 0* arise.

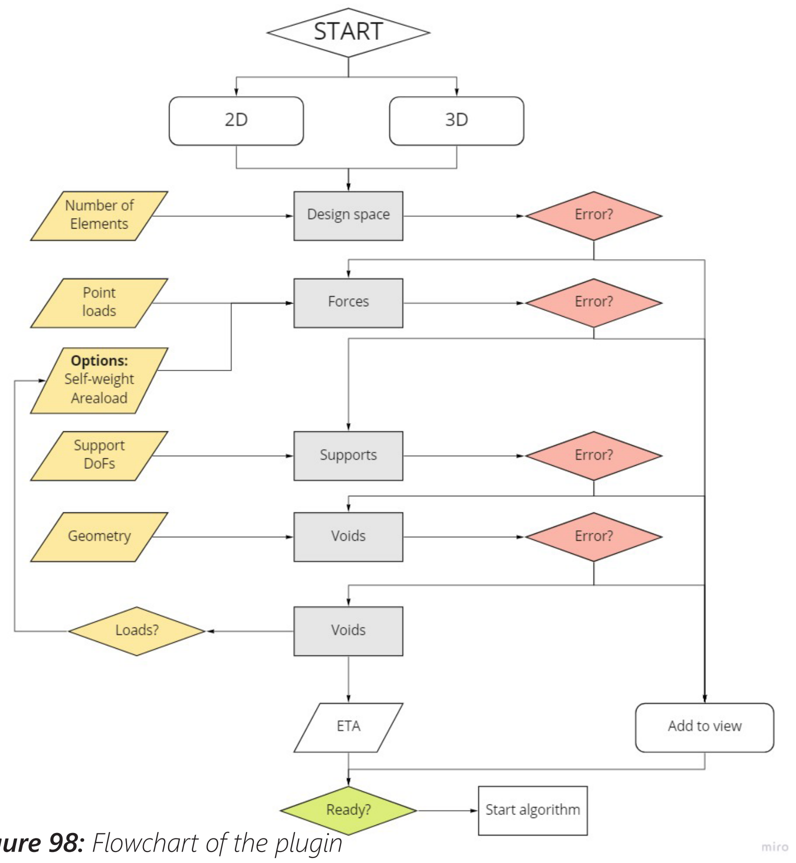


Figure 98: *Flowchart of the plugin*

Another output of the solver should be an estimated time that it takes to solve this problem. Chapter 3.8 showed already the hefty calculation times that happen, especially with 3D problems. The model viewer could include 1 loop of the solver and check the time of this one loop. Assuming the solver will take around maxloops loops, the ETA can be calculated. Including 1 loop to solve in the model can also be very beneficial in the showing of any errors. The ETA is based on the values that were found in chapter 3.8.

Giving errors
Showing errors early on is important and allows for users to solve these errors more intuitively. Most errors will happen because of poor placements of the inputs and can be showed easily to the user. The sys library is used to output any errors as a text in the errors output of the component. When no errors arise in the input, one loop is performed in order to see if the first FEA is performed correctly. This loop is calculated using sp.solve, in order to ignore the longer calculation times.

Errors will most likely only happen with wrongly placed forces, supports and voids. These erros can be coded into the model viewer, as the point that will represent a force or support should always be within or on the edge of the design space.

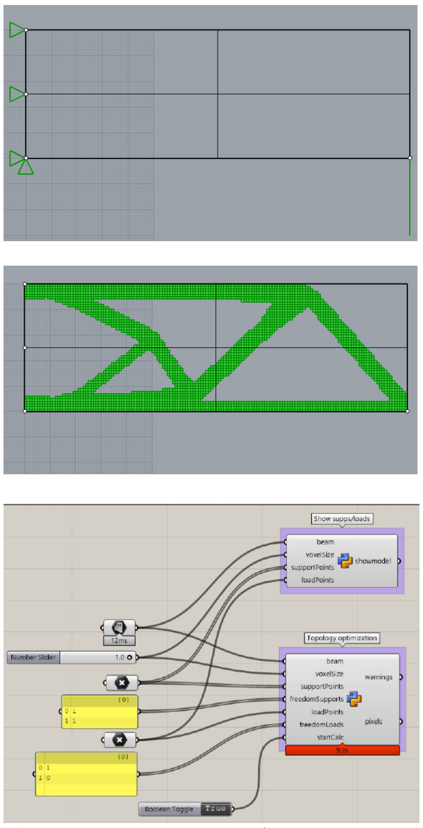


Figure 99: *TO in Grasshopper*

Translating geometry into matrices
Toy problem 1 included already a way to translate Rhino geometry into matrices and toy problem 5 did that for 3D geometry. As these are very different procedures and different codes, two different components are designed. In 2D, the design space is given as a surface, the voids as surfaces or lines, the support and forces as points, with corresponding directions. Then, some standard choices are included which give a offer more problems to be solved. These options are:

- Self-weight
- Roof area-load
- Ground as foundation
- Voids have weight

The 3D model creator is a bit more complex to create, but in essence works the same. A mesh or Brep is imported as the design space and voids. Supports and forces are a bit harder, as they should be able to be imported as points and surfaces, as a preset load or area load. The model allows for both to happen, although two different inputs have to be created in order to function well. The model viewer will show the model as shown in figure 99, unless errors occur. It is important that errors are found in this stage, so that very little errors occur in the solving part.

Translating the solved system in geometry
As the voxelSize is given as an input and the design space always starts at the coordinate (0,0), translating the solved system is not that hard to do. Rhino will look at each element in x and compare it to a certain threshold. When the value of x is larger, it is placed with its correct position and size. The following pseudocode describes that for 3D:

```
counter = 0
grid = []
for elex,eley,elez in rangesOf(nelx,nely,nelz):
    if x[counter]>y:
        point = Point(nelx-elex-1,eley,
                                        nelz - elez - 1)
        point2 = Point(nelx- elex, eley+1,
                                        nelz - elez)
        grid.add(BoxFromDiag(point, point2)
    counter = counter + 1
```

Grasshopper implementation
The implementation in Grasshopper is important in order to distribute knowledge about topology optimization around architecture students. Therefore, the creation of this plugin will be definitely performed, but falls outside the timeframe and scope of this research. Implementing heavy algorithms like this in Grasshopper is hard, risky and time consuming. Finding an alternative to the Proxy method to implement NumPy in Grasshopper is probably essential to run this algorithm fluently.

Nevertheless, the plugin will be developed and this research should be the basis of that plugin. It should allow for intuitive use in order to generated simple, but optimized geometry.

0 4 C o n c l u s i o n s

4.1 Conclusions

The objective of this research was to *implement design dependent loads in topology optimization and apply this algorithm to buildings*. In order to fulfill this objective, several sub-objectives are created and those are divided into 2 smaller toy problems. Using these toy problems will systematic solve the problems that occur. Each problem had their own objectives stated and this allows for a systematic approach. Using this approach will also allow for a lot of feedback during each step of the process.

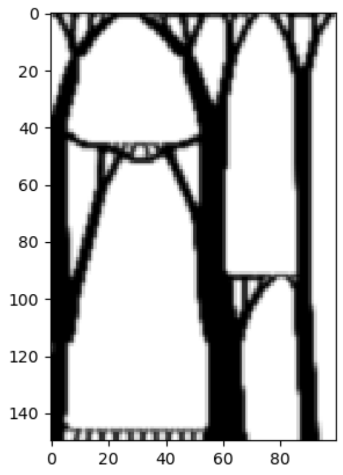


Figure 100: *Section of a building*

The first sub-objective was to create an algorithm that solves problems using topology optimization. To achieve this sub-objective, two toy problems were created. The first problem asked for an integration in Grasshopper and a simple tunnel as a problem. The implementation in Grasshopper was written and allowed for the first testing of the design. Using the Rhino geometry, the tunnel could be replicated and solved. To solve the system, known topology optimization codes by Deetman and Sigmund were edited, so the translation to Grasshopper could be made easier. Also the implementation of voids was added and the possibility of more complex design spaces. Figure 100 shows the solution for the second toy problem, which extended the possiblities of the algorithm, using more complex design spaces and multiple voids. Verifying this topology optimized shape with a program like Ansys shows that this algorithm behaves properly according to the constraints.

The second sub-objective was to *implement design dependent loads in the algorithm*, to make it more applicable in the architectural field. In architectural models, selfweight is very important, as the weight of the construction is usually the largest force on the design. This was implemented using the mathematical desciption by Duysinx and Bruyneel. Usually in architecture there are no direct forces on the building, which is a problem as topology optimization needs some kind of force to optimize. To counter this a roofconstraint was added, which told the optimizer to only accept solutions with a roof over all the voids. Lastly, a snowload was added, as this force will fall on the roof, dependent on its shape. Without these forces the optimizer will create thin shapes, in order to minimize the selfweight forces. Combining all these additions resulted in the arch that is shown in figure 101. Verification in Ansys showed that the geometry is quite strong, deforming very slightly when put under the forces.

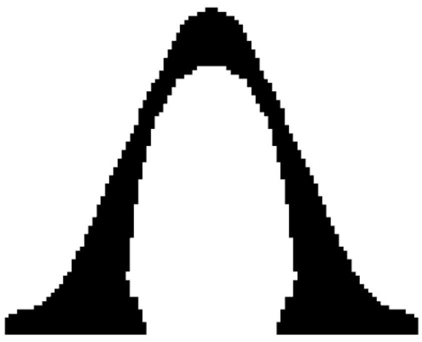


Figure 101: *Arch under selfweight and snowload*

Lastly for the last sub-objective the algorithm and the methodology were *translated into the 3D geometry*. The largest benefit in this translation is that most calculations are done with vectors. Translating between the 3D arrays and the 1D arrays was the main issue. Implementing a methodology created by Liu and Tovar was the solution for most of the issues. Also selfweight, snowloads and the roof constraint were implemented in the 3D algorithm which set the basis to discover the possibilities that are available within topology optimization. Figure XX shows the final result of all the solved problems that were met using the toy problems.

Topology optimization showed itself to be a great tool for the generating of shapes. It allows for very specific implementations of the needed design, such as selfweight and complex design spaces. As topology optimization uses MMA as a solver, adding several constraints can push the design in a certain direction, without influencing the design too much. The translation to 3D is easily made, which is very convenient in its application within architecture.

The main problem with topology optimization is its computational time. Generating high-resolution 3D geometry is very costly and makes the implementation in Grasshopper nearly impossible. Choosing a good solver could save a lot of time, using the conjugate gradient method with good preconditioners with large systems. Smaller systems are much quicker calculated using the standard solver. Very simple systems benefit from taking the Optimality Criteria in stead of the MMA.

The main reseach question was *"How can we design structures for masonry buildings using topology optimization?"*. Topology optimization can be a great tool for early structural form finding. It allows for intuitive design, generating geometry following a predefined system. The implementation of selfweight allows this to be implemented in masonry buildings. The next chapter will shortly discuss possible applications.

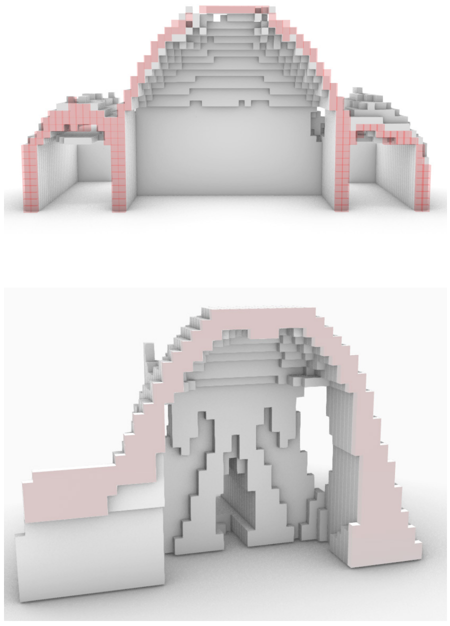


Figure 102: *Results of the translation to 3D*

4.2 Applicability

This chapter will answer the last research question *"How can topology optimization transform the design process of masonry buildings, particularly their configuring and shaping processes?"*. This thesis will not give a final answer on this question, but this chapter will reflect on this question.

The algorithm that is created in this thesis is in the current state not usable for final designs of buildings, but the focus should be placed on the early structural form finding. As the focus of this research is mainly towards architecture, the 3D part of the algorithm is much more interesting and applicable.

A futuristic approach of configuring and shaping masonry buildings would be to import several inputs, like voids, supports and forces and optimize its shape. This shape being dependent on the brick that one is using to build with. If the shape of the brick could be used as input in the optimization process, the shaping of geometry could be very efficient and applicable to most masonry buildings.

In this futuristic approach I think topology optimization is the core foundation on how masonry buildings could be configured and shaped. Topology optimization can be made very intuitive, as most inputs are basic vectors. Several constraints can be added which allow for specific results. Further developments in topology optimization could include the translation to from voxels to bricks, multi-material or the optimal locations of elements.

This research shows that geometry can be shaped and that it follows our expectation on how geometry would eventually look like. Arches and domes are shaped, which are basic design elements in masonry architecture. This shows that it topology optimization could be used perfectly for masonry architecture.

The main issue with topology optimization is its calculation time.
For early structural form-finding a low resolution is sufficient, but to generate more detailed geometry, a higher resolution is needed. In 3D increasing the resolution will exponentially increase its computational time. For plugins this is very unusable, but in comparison with the time it takes a human to design geometry, it might be very fast. Developing a methodology to configure the geometry is essential to solve problems with a very high resolution.

In conclusion, I think topology optimization can be the foundation of a change in how masonry buildings are shaped. It has many possibilities and can be a very intuitive process. The main necessity is the development of a methodology in order to retrieve high resolution and useful results.

05 References

>>>5.1 Literature

Profiling Python using cProfile: a concrete case. (n.d.). Retrieved June 28, 2020, from https://julien.danjou.info/guide-to-python-profiling-cprofile-concrete-case-carbonara/

scipy.sparse.linalg.cg — SciPy v1.5.0 Reference Guide. (n.d.). Retrieved June 29, 2020, from https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.cg.html#scipy.sparse.linalg.cg

Avdić, I. (2019). *BIO-INSPIRED APPROACH TO EARLY STAGE STRUCTURAL FORM FINDING Faculty of Architecture and the Built Environment*. https://doi.org/uuid:ebed5ec7-7951-4139-b80f-eba8052c86c1

Bathe, K. J. (2006). Finite element procedures. Second edition. In *Mit*. http://web.mit.edu/kjb/www/Books/FEP_2nd_Edition_4th_Printing.pdf

Bendsøe, M. P., & Sigmund, O. (2004). Topology Optimization. In *Topology Optimization*. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-05086-6

Bruyneel, M., & Duysinx, P. (1997). *Topology Optimization With Self-Weight Loading: Unexpected Problems and Solutions*. *4*, 1–4. https://www.researchgate.net/publication/277093663_Topology_optimization_with_self-weight_loading_un-expected_problems_and_solutions

Bruyneel, M., & Duysinx, · P. (n.d.). *Struct Multidisc Optim (2005) 29: 245-256 Note on topology optimization of continuum structures including self-weight*. https://doi.org/10.1007/s00158-004-0484-y

Bruyneel, M., & Duysinx, P. (2005). Note on topology optimization of continuum structures including self-weight. *Structural and Multidisciplinary Optimization*, *29*(4), 245–256. https://doi.org/10.1007/s00158-004-0484-y

Cai, K. (2011). A simple approach to find optimal topology of a continuum with tension-only or compression-only material. *Structural and Multidisciplinary Optimization*, *43*(6), 827–835. https://doi.org/10.1007/s00158-010-0614-7

Finite, T., & Previous, E. (2016). *4.1.3 Shape Function*. 13–16. https://www.iue.tuwien.ac.at/phd/orio/node48.html

Huang, X., & Xie, Y. M. (2009). Bi-directional evolutionary topology optimization of continuum structures with one or multiple materials. *Computational Mechanics*, *43*(3), 393–401. https://doi.org/10.1007/s00466-008-0312-0

Jain, N., & Saxena, R. (2018). Effect of self-weight on topological optimization of static loading structures. *Alexandria Engineering Journal*, *57*(2), 527–535. https://doi.org/10.1016/j.aej.2017.01.006

Jain, N., & Saxena, R. (2018). Effect of self-weight on topological optimization of static loading structures. *Alexandria Engineering Journal*, *57*(2), 527–535. https://doi.org/10.1016/j.aej.2017.01.006

John, G., Clements-Croome, D., & Jeronimidis, G. (2005). Sustainable building solutions: A review of lessons from the natural world. *Building and Environment*, *40*(3), 319–328. https://doi.org/10.1016/j.buildenv.2004.05.011

Langelaar, M. (2020) (Personal communication, June 22th 2020)

Liu, K., & Tovar, A. (2014). An efficient 3D topology optimization code written in Matlab. *Structural and Multidisciplinary Optimization*, *50*(6), 1175–1196. https://doi.org/10.1007/s00158-014-1107-x

Martins, J. R. R. A., & Poon, N. M. K. (n.d.). *6 th World Congress on Structural and Multidisciplinary Optimization On Structural Optimization Using Constraint Aggregation*.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Saad, Y., & Schultz, M. H. (1986). GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, *7*(3), 856–869. https://doi.org/10.1137/0907058

Shewchuk, J. R. (1994). Conjugate Gradient Method Without the Agonizing Pain. *Science*.

Sigmund, O. (2001). A 99 line topology optimization code written in matlab. *Structural and Multidisciplinary Optimization*, *21*(2), 120–127. https://doi.org/10.1007/s001580050176

Sleijpen, G., & Van Gijzen, M. (2017). *National Master Course Numerical Linear Algebra Improving iterative solvers: preconditioning, deflation, numerical software and parallelisation*.

Svanberg, K. (n.d.). *The Method of Moving Asymptotes-Modelling aspects and solution schemes*.

Wang, X. Y., Huang, T. Z., & Deng, L. J. (2018). Single image super-resolution based on approximated Heaviside functions and iterative refinement. *PLoS ONE*, *13*(1). https://doi.org/10.1371/journal.pone.0182240

Zhang, J., Zhang, W. H., Zhu, J. H., & Xia, L. (2012). Integrated layout design of multi-component systems using XFEM and analytical sensitivity analysis. *Computer Methods in Applied Mechanics and Engineering*, *245–246*, 75–89. https://doi.org/10.1016/j.cma.2012.06.022

Zuo, Z. H., & Xie, Y. M. (2015). A simple and compact Python code for complex 3D topology optimization. *Advances in Engineering Software*, *85*, 1–11. https://doi.org/10.1016/j.advengsoft.2015.02.006

>>>5.2 Equations

This will list all the equations that have a source:

Bathe, 2006
[2, 5, 8]

Bensoe & Sigmund
[6, 7]

Bruyneel & Duysinx, 2005
[18, 19]

Huang & Deng, 2018:
[22, 33]

Langelaar, 2020
[13, 14, 15, 16, 17, 24, 25, 26, 28, 29, 30, 31, 32, 37, 38, 39]

Martins & Roon, 2005
[27]

Sigmund, 2001:
[0, 1, 5, 8, 9, 10, 11]

Sleijpen & van Gijzen, 2017
[41, 42, 43]

Author
[3, 4, 12, 20, 21, 34, 35, 36, 40]

>>>5.3 Figures

The following figures contain references to the figure, unreferences figures are made by the author.

1: Wienerberger building solutions (2020). Retrieved 20 February 2020 from https://www.wienerberger-building-solutions.com/Expertise/Our-expertise/Benefits-of-building-with-clay.html.

2: Tony Abbey (2017). Retrieved 20 February 2020 from https://www.digitalengineering247.com/article/topology-optimization/.

3: QNCC (2019). Retrieved 20 February 2020 from http://www.qncc.com/site/en/Newsletters/The_Sidra_Signature.aspx

7: Avdić, I. (2019)

9: Sigmund, O. (2001)

11: Avdić, I. (2019)

16: De Orio (2008)

20: Geeks for Geeks (2019), retrieved at 4 May 2020 from https://www.geeksforgeeks.org/program-for-bisection-method/

23: Blackman & Miller (2014)

26: Liu and Tovar (2014)

27: Liu and Tovar (2014)

31: House Design Coffee (2012), retrieved at 20 February 2020 from https://www.house-design-coffee.com/catenary-arch.html

69: Naboni, Roberto (2018). - Architectural Morphogenesis Through Topology Optimization

80: Avdić, I. (2019)

88: Archdaily (2017). - AD Classics, Haus am Horn, retrieved at 26 June 2020 from https://www.archdaily.com/873082/ad-classics-haus-am-horn-germany-georg-muche

Appendix A: 2D Topology optimization code

02/07/2020

1 2D Code:

```python

# A 165 LINE TOPOLOGY OPTIMIZATION CODE BY NIELS AAGE AND VILLADS EGEDE JOHANSEN, JANUARY 2013
# MMA OPTIMIZER ADDED BY ARJEN DEETMAN, NOVEMBER 2019
# Architectural implementations by Rick van Dijk, July 2020

from __future__ import division
import numpy as np
from scipy.sparse import coo_matrix
from scipy.sparse.linalg import spsolve
from scipy.sparse.linalg import cg
from matplotlib import colors
import matplotlib.pyplot as plt
from scipy.sparse import diags
from scipy.linalg import solve

# MAIN DRIVER
def main(nelx,nely,volfrac,penal,rmin,ft,xsolv,sw,snow,roofc):
    ne = nelx * nely
    # Max and min stiffness
    Emin = 1e-9
    Emax = 1.0
    # dofs:
    ndof = 2*(nelx+1)*(nely+1)
    # Allocate design variables (as array), initialize and allocate sens.
    n = nely*nelx
    nele = n
    x = volfrac*np.ones(nely*nelx, dtype=float)
    xPhys = x.copy()
    dc = np.zeros((nely,nelx), dtype=float)
    # Initialize OC
    if xsolv == 0:
        xold1 = x.copy()
        g = 0 # must be initialized to use the NGuyen/Paulino OC approach
    # Initialize MMA
    elif xsolv == 1:
        m = 1
        if roofc == True:
            m = 2
        xmin = np.zeros((n,1))
        xmax = np.ones((n,1))
        xval = x[np.newaxis].T
        xold1 = xval.copy()
        xold2 = xval.copy()
        low = np.ones((n,1))
        upp = np.ones((n,1))
        a0 = 1.0
        a = np.zeros((m,1))
        c = 10000*np.ones((m,1))
        d = np.zeros((m,1))
        move = 0.2
    # FE: Build the index vectors for the for coo matrix format.
    KE = lk()
    edofMat = np.zeros((nelx*nely,8),dtype=int)
    for elx in range(nelx):
        for ely in range(nely):
            el = ely+elx*nely
            n1 = (nely+1)*elx+ely
            n2 = (nely+1)*(elx+1)+ely
            edofMat[el,:] = np.array([2*n1+2, 2*n1+3, 2*n2+2, 2*n2+3,2*n2, 2*n2+1, 2*n1, 2*n1+1])
    # Construct the index pointers for the coo format
    iK = np.kron(edofMat,np.ones((8,1))).flatten()
    jK = np.kron(edofMat,np.ones((1,8))).flatten()
    vertedof = edofMat.reshape(-1)[0::2].reshape((nelx*nely, 4)) + 1  #Get the vertical dofs
    # Filter: Build (and assemble) the index+data vectors for the coo matrix format
    nfilter = int(nelx*nely*((2*(np.ceil(rmin)-1)+1)**2))
    iH = np.zeros(nfilter)
    jH = np.zeros(nfilter)
    sH = np.zeros(nfilter)
    cc = 0

    for i in range(nelx):
        for j in range(nely):
            row = i*nely+j
            kk1 = int(np.maximum(i-(np.ceil(rmin)-1),0))
            kk2 = int(np.minimum(i+np.ceil(rmin),nelx))
            ll1 = int(np.maximum(j-(np.ceil(rmin)-1),0))
            ll2 = int(np.minimum(j+np.ceil(rmin),nely))
            for k in range(kk1,kk2):
                for l in range(ll1,ll2):
                    col = k*nely+l
                    fac = rmin-np.sqrt(((i-k)*(i-k)+(j-l)*(j-l)))
                    iH[cc] = row
                    jH[cc] = col
                    sH[cc] = np.maximum(0.0,fac)
                    cc = cc+1

    # Finalize assembly and convert to csc format
    H = coo_matrix((sH,(iH,jH)),shape=(nelx*nely,nelx*nely)).tocsc()
    Hs = H.sum(1)

    # SUPPORTS
    dofs = np.arange(2*(nelx+1)*(nely+1))
    fixed = []
    #Set floor as supports
    counter = 0
    for element1 in range(0,nelx+1,1):
        node = int(element1 * (nely +1) + nely)
        counter = counter + 1
        fixed.append(2*node)
        fixed.append(2*node+1)

    #fixed = [supp1, supp2, supp3,.... suppn]
    free = np.setdiff1d(dofs,fixed)

    #VOIDS
    voids = np.zeros((nele))
    #Get the voids from the input model in Rhino

    """
    voidlist = [void1ID, void2ID, void3ID .... voidnID]
    voids[voidlist] = 1
    """

    #Update the volfraction according to the amount of voids
    volfrac = volfrac*(nele-len(voidlist)) / (nele)

    # Solution and RHS vectors
    f = np.zeros((ndof))
    u = np.zeros((ndof,1))
    # Set loop counter and gradient vectors
    loop = 0
    change = 1
    dv = np.ones(nely*nelx)
    dc = np.ones(nely*nelx)
    ce = np.ones(nely*nelx)

    fact1 = .01 #Factors of the roofconstraint
    fact2 = .01 #Factor of the senstivity of the roofconstraint

    #Create the dFdx vector
    swFactor = 16
    selfweightvector = np.zeros(ndof)
    selfweightvector[1::2] = swFactor/ (volfrac * nely * nelx) / 4

    testtt = np.zeros(n)
    uold = np.zeros(ndof)
    dcSH = np.zeros((nelx,nely))
    dcsnow = np.zeros((ndof))

    while (change>0.001) and (loop<500):
        #Step 1 is to implement self-weight
        if sw == True:
            currIndex = 0
            x.reshape((n))

            for xi in x:
                totalself =swFactor/ (volfrac * nely * nelx)
                selfweight = xi * totalself / 4 #some factor
                if selfweight>totalself/swFactor:
                    currDoF = vertedof[currIndex]
                    f[currDoF] += selfweight
                currIndex += 1

            #or use the following:
            """
            totalself = 8 / (volfrac * nely * nelx)/4
            selfstep = 0.5 + np.arctan((x - 0.4)/0.0001)/ np.pi
            selfweights = totalself * selfstep
            f[vertedof] += np.tile(selfweights,(4,1)).T
            """

        ####PRESET F:
        f = f.reshape(-1)
        #f[indexF] = 1

        #DeadWeight on the Roof
        """
        for roofelement in range(nelx):
            topnodeleft = roofelement*(2*(nely+1)) + 1
            topnoderight = (roofelement+1)*(2*(nely+1)) + 1
            f[topnodeleft] = 0.01
            f[topnoderight] = 0.01
        """

        #Set a random Force (uneven = vertical)
        #f[[10363]]+=1

        #Apply snowload
        if snow == True:
            snowp = 1
            x2 = np.arctan((x - 0.2)/0.0001)/np.pi
            cumx = np.subtract(np.cumsum(np.power(x2.reshape(nelx,
    nely),snowp),axis=1),(np.power(x2.reshape(nelx,nely),snowp)))
            downstepcumx = np.cumsum(np.power(x2.reshape(nelx,nely)
    ,snowp),axis=1)
            SH = np.arctan((downstepcumx - 0.5)/0.01)/np.pi - np.
    arctan((downstepcumx - 1.5)/0.01)/np.pi
            snowForce = (0.001 / 4) * SH
            f[vertedof] += np.tile(snowForce.reshape(n),(4,1)).T


        # Setup and solve FE problem
        sK = ((KE.flatten()[np.newaxis]).T*(Emin+(xPhys)**penal*(
    Emax-Emin))).flatten(order='F')
        K = coo_matrix((sK,(iK,jK)),shape=(ndof,ndof)).tocsc()
        # Remove constrained dofs from matrix
        K = K[free,:][:,free]

        # Solve system with solver dependent on shape
        if nele < 10000:
            u[free,0] = spsolve(K,f[free,0])
        else:
            xguess = u
            resultcgkg = cg(K,f[free], x0=xguess[free],tol=1e-04, M
    = diags(np.divide(1,K.diagonal())))
            u[free,0]= resultcgkg[0]

        # Objective and sensitivities
        ce[:] = (np.dot(u[edofMat].reshape(nelx*nely,8),KE) * u[edofMat].reshape(nelx*nely,8)).sum(1)

        # Configure snow sensitivities
        if snow == True:
            dydx = np.where(np.power(x2.reshape(nelx,nely),snowp)>cumx, 1,0)

            dcSH1 = 100 / (np.pi*(10000*np.power((downstepcumx-0.5),2)+1))
            dcSH2 = 100 / (np.pi*(10000*np.power((downstepcumx-1.5),2)+1))
            dcSH = dcSH1 - dcSH2

            dcsnowload = ((0.005 / 4) * dcSH * dydx).reshape(-1)
            dcsnow[vertedof] += np.tile(dcsnowload.reshape(n),(4,1)).T

        dcselfweight = np.sum(np.squeeze(u[edofMat]) * np.squeeze((selfweightvector+dcsnow)[edofMat]) * 2,1)
        ceselfweight = np.sum(np.squeeze(u[edofMat])* np.squeeze(f[edofMat]) * 2,1)

        #Calculate compliance & sensitivity
        if sw == True:
            obj = ((Emin+xPhys**penal*(Emax-Emin))*ce + ceselfweight).sum()
            dc[:] = ((-penal*xPhys**(penal-1)*(Emax-Emin))*ce + dcselfweight)
        else:
            obj = ((Emin+xPhys**penal*(Emax-Emin))*ce).sum()
            dc[:] = ((-penal*xPhys**(penal-1)*(Emax-Emin))*ce)
        dv[:] = np.ones(nely*nelx)

        #Roofing constraint
        if roofc == True:
            voidexistin = np.where(np.sum(voids.reshape(nelx,nely), axis=1))    #Output are the column indeces with voids in it
            invertedarea = 1 - voids.reshape(nelx,nely)[voidexistin]         #Inverting the area of these indeces, in order to multiply

            xroofconstraint = x.reshape(nelx,nely)[voidexistin]         #Get the x-values at these columns

            epgk = np.exp(10*(1-np.sum(np.power(xroofconstraint,5), axis=1)))
            gcolumns = np.log(np.sum(epgk))/10
            dcrc = epgk / gcolumns
            dcrcc = np.zeros((nelx,nely))
            dcrcc[voidexistin] = np.multiply(np.tile(dcrc,(nely,1)),invertedarea.T).T   #Sensitivity of the constraint

        # Sensitivity filtering:
        if ft == 0:
            dc[:] = np.asarray((H*(x*dc))[np.newaxis].T/Hs)[:,0] / np.maximum(0.001,x)
        elif ft == 1:
            dc[:] = np.asarray(H*(dc[np.newaxis].T/Hs))[:,0]
            dv[:] = np.asarray(H*(dv[np.newaxis].T/Hs))[:,0]

        #Update x according to the existance of voids
        x = np.where(voids, 0.001, x)

        # Optimality criteria
        if xsolv == 0:
            xold1[:] = x
            (x[:],g) = oc(nelx,nely,x,volfrac,dc,dv,g)
        # Method of moving asymptotes
        elif xsolv == 1:
            mu0 = 0.1 # Scale factor for objective function
            mu1 = 0.1 # Scale factor for volume constraint function
            f0val = mu0*obj
            df0dx = mu0*dc[np.newaxis].T
            volumecons = np.squeeze(mu1*np.array([[xPhys.sum()/n-volfrac]]))
            dxvolume = mu1*(dv/(n*volfrac))[np.newaxis]
            if roofc == True:
                gval = gcolumns*fact1 #0.0001
                fval = np.array([[volumecons] ,[gval]]) #Constraint naar een vector maken ->fval= [volumns, gval]
                dxroof = (dcrcc.reshape(-1)*fact2).reshape(1,n)
                dfdx = np.concatenate((dxvolume,-dxroof))
            else:
                fval = volumecons
                dfdx = dxvolume
            xval = x.copy()[np.newaxis].T

            xmma,ymma,zmma,lam,xsi,eta,mu,zet,s,low,upp = \
                mmasub(m,n,k,xval,xmin,xmax,xold1,xold2,f0val,df0dx,fval,dfdx,low,upp,a0,a,c,d,move)

            xold2 = xold1.copy()
            xold1 = xval.copy()
            x = xmma.copy().flatten()
        # Filter design variables
        if ft == 0: xPhys[:] = x
        elif ft == 1: xPhys[:] = np.asarray(H*x[np.newaxis].T/Hs)[:,0]
        # Compute the change by the inf. norm
        change = np.linalg.norm(x.reshape(nelx*nely,1)-xold1.reshape(nelx*nely,1),np.inf)
        # Write iteration history to screen (req. Python 2.6 or newer)
        loop = loop+1
        print("it.: {0} , obj.: {1:.3f} Vol.: {2:.3f}, ch.: {3:.3f}".format(loop,obj,x.sum()/n,change))

        #Reset all the forces
        f = np.zeros(ndof)


    # Plot result in the dirName folder
    fig,ax = plt.subplots()
    im = ax.imshow(-xPhys.reshape(nelx,nely).T, cmap='gray',
    interpolation='none', norm=colors.Normalize(vmin=-1,vmax=0))
    plt.savefig(dirName + '/loop' + str(loop)+'obj' + str(obj)+'.
    png')
    np.savetxt(dirName+"/testingswv.csv", x , delimiter=",")

#element stiffness matrix
def lk():
    E = 1
    nu = 0.3
    k = np.array([1/2-nu/6,1/8+nu/8,-1/4-nu/12,-1/8+3*nu/8,-1/4+nu
    /12,-1/8-nu/8,nu/6,1/8-3*nu/8])
    KE = E/(1-nu**2)*np.array([ [k[0], k[1], k[2], k[3], k[4], k
    [5], k[6], k[7]],
    [k[1], k[0], k[7], k[6], k[5], k[4], k[3], k[2]],
    [k[2], k[7], k[0], k[5], k[6], k[3], k[4], k[1]],
    [k[3], k[6], k[5], k[0], k[7], k[2], k[1], k[4]],
    [k[4], k[5], k[6], k[7], k[0], k[1], k[2], k[3]],
    [k[5], k[4], k[3], k[2], k[1], k[0], k[7], k[6]],
    [k[6], k[3], k[4], k[1], k[2], k[7], k[0], k[5]],
    [k[7], k[2], k[1], k[4], k[3], k[6], k[5], k[0]] ]);
    return (KE)

def oc(nelx,nely,x,volfrac,dc,dv,g)... #See Deetman(2020)

def subsolv(m,n,epsimin,low,upp,alfa,beta,p0,q0,P,Q,a0,a,b,c,d):
    ... #See Deetman(2020)

def mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2,f0val,df0dx,fval,
    dfdx,low,upp,a0,a,c,d,move):... #See Deetman
```
Listing 1: 2D Topology optimization code

Appendix B: 3D Topology optimization code

02/07/2020

1 2D Code:

```

#use python interpretor that uses python2.7
#Translation of Liu and Tovar (2014) by Rick van Dijk
#Implementation of selfweight, roofconstrant in 3D TopOp

import numpy as np
import scipy
from scipy import sparse
from scipy.sparse import linalg
from scipy.sparse import diags # or use numpy: from numpy import
    diag as diags
from scipy.linalg import solve # or use numpy: from numpy.linalg
    import solve
from matplotlib import colors
import matplotlib.pyplot as plt

def oc(nelx,nely,x,volfrac,dc,dv,g)... #See Deetman(2020)

def subsolv(m,n,epsimin,low,upp,alfa,beta,p0,q0,P,Q,a0,a,b,c,d):
    ... #See Deetman(2020)

def mmasub(m,n,iter,xval,xmin,xmax,xold1,xold2,f0val,df0dx,fval,
    dfdx,low,upp,a0,a,c,d,move):... #See Deetman



##Inputs
nelx = 30
nely = 20
nelz = 40

volfrac = 0.3
penal = 3
rmin = 1.5

sw = True #sw==True -> selfweight will be applied
snow = False #snow==True -> snowload will be applied
roofc = True #roofc == True -> roofconstraint will be applied

maxloop = 100
tolx = 0.001
```

```python
displayflag = 0

##USER-DEFINED MATERIAL PROPERTIES
E0 = 1
Emin = 1e-9
nu = 0.3
nele = nelx*nely*nelz
n = nele
ndof = 3*(nelx+1)*(nely+1)*(nelz+1)
##USER-DEFINED SUPPORT FIXED DOFs

fixed = []

for element1 in range(0,nelx+1,1):
    for element2 in range(0,nelz+1,1):
        node = np.multiply(element2, (nelx + 1)*(nely + 1)) + np.
    multiply(element1, nely + 1) + np.add(np.negative(0),nely)
        fixed.append(node)

fixeddof = [3*np.reshape(fixed,-1)+1,3*np.reshape(fixed,-1),3*np.
    reshape(fixed,-1)-1]

#VOIDS
voids = np.zeros((nele))
#Get the voids from the input model in Rhino

"""
voidlist = [void1ID, void2ID, void3ID .... voidnID]
voids[voidlist] = 1
volfrac = volfrac*(nele-len(voidlist)) / (nele)
"""

##PREPARE FINITE ELEMENT ANALYSIS
U = np.zeros((ndof))
freedofs = np.setdiff1d(np.arange(1,ndof,1),fixeddof)

KE = []

A = np.array([[32, 6, -8, 6, -6, 4, 3, -6, -10, 3, -3, -3, -4, -8],
     [-48, 0, 0, -24, 24, 0, 0, 0, 12, -12, 0, 12, 12, 12]], dtype
    = np.float32)
k = (np.transpose(A[0])*1 + np.transpose(A[1])*0.3) / 144

K1 = np.array([[k[0], k[1], k[1], k[2], k[4], k[4]],
    [k[1], k[0], k[1], k[3], k[5], k[6]],
    [k[1], k[1], k[0], k[3], k[6], k[5]],
    [k[2], k[3], k[3], k[0], k[7], k[7]],
    [k[4], k[5], k[6], k[7], k[0], k[1]],
    [k[4], k[6], k[5], k[7], k[1], k[0]]])

K2 = np.array([[k[8],  k[7],  k[11], k[5],  k[3],  k[6]],
    [k[7],  k[8],  k[11], k[4],  k[2],  k[4]],
    [k[9], k[9], k[12], k[6],  k[3],  k[5]],
    [k[5],  k[4],  k[10], k[8],  k[1],  k[9]],
    [k[3],  k[2],  k[4],  k[1],  k[8],  k[11]],
    [k[10], k[3],  k[5],  k[11], k[9], k[12]]])

```

```
K3 = np.array([[k[5],  k[6],  k[3],  k[8],  k[11], k[7]],
    [k[6],  k[5],  k[3],  k[9], k[12], k[9]],
    [k[4],  k[4],  k[2],  k[7],  k[11], k[8]],
    [k[8],  k[9], k[1],  k[5],  k[10], k[4]],
    [k[11], k[12], k[9], k[10], k[5],  k[3]],
    [k[1],  k[11], k[8],  k[3],  k[4],  k[2]]])

K4 = np.array([[k[13], k[10], k[10], k[12], k[9], k[9]],
    [k[10], k[13], k[10], k[11], k[8],  k[7]],
    [k[10], k[10], k[13], k[11], k[7],  k[8]],
    [k[12], k[11], k[11], k[13], k[6],  k[6]],
    [k[9], k[8],  k[7],  k[6],  k[13], k[10]],
    [k[9], k[7],  k[8],  k[6],  k[10], k[13]]])

K5 = np.array([[k[0], k[1],  k[7],  k[2], k[4],  k[3]],
    [k[1], k[0],  k[7],  k[3], k[5],  k[10]],
    [k[7], k[7],  k[0],  k[4], k[10], k[5]],
    [k[2], k[3],  k[4],  k[0], k[7],  k[1]],
    [k[4], k[5],  k[10], k[7], k[0],  k[7]],
    [k[3], k[10], k[5],  k[1], k[7],  k[0]]])

K6 = np.array([[k[13], k[10], k[6],  k[12], k[9], k[11]],
    [k[10], k[13], k[6],  k[11], k[8],  k[1]],
    [k[6],  k[6],  k[13], k[9], k[1],  k[8]],
    [k[12], k[11], k[9], k[13], k[6],  k[10]],
    [k[9], k[8],  k[1],  k[6],  k[13], k[6]],
    [k[11], k[1],  k[8],  k[10], k[6],  k[13]]])

stack = np.vstack([np.hstack([K1, K2, K3, K4]),np.hstack([np.
    transpose(K2), K5, K6, np.transpose(K3)]),np.hstack([np.
    transpose(K3), K6, np.transpose(K5), np.transpose(K2)]),np.
    hstack([K4, K3, K2, np.transpose(K1)])])
KE = 1/((nu+1)*(1-2*nu)) *  stack

nodegrd = np.reshape(np.arange(0,(nely+1)*(nelx+1),1), [nelx+1,nely
    +1])
newitems = np.transpose(nodegrd[:-1])[:-1].transpose()
nodeids = np.reshape(newitems,[nely*nelx,1])
nodeidz = np.arange(0,(nelz-1)*(nely+1)*(nelx+1)+1,(nely+1)*(nelx
    +1))

nodeids = np.repeat(nodeids, len(nodeidz), axis=1) + np.tile(
    nodeidz, len(nodeids)).reshape(((nelx*nely),nelz))+1
edofVec = (3 * np.reshape(np.transpose(nodeids),-1)+1)

repeatedVec = np.repeat(edofVec, 24, axis=0).reshape(len(edofVec)
    ,24)

repeatingMatrix = [
    0,1,2,(3*nely+3),
    (3*nely+4),(3*nely+5),(3*nely+0),(3*nely+1),
    (3*nely+2),-3,-2,-1,
    (3*(nely+1)*(nelx+1)+0),(3*(nely+1)*(nelx+1)+1),(3*(nely+1)*(
    nelx+1)+2),(3*(nely+1)*(nelx+1)+3*nely + 3),
    (3*(nely+1)*(nelx+1)+3*nely + 4),(3*(nely+1)*(nelx+1)+3*nely +
    5),(3*(nely+1)*(nelx+1)+3*nely + 0),(3*(nely+1)*(nelx+1)+3*nely
     + 1),
    (3*(nely+1)*(nelx+1)+3*nely + 2),(3*(nely+1)*(nelx+1)-3),(3*(
    nely+1)*(nelx+1)-2),(3*(nely+1)*(nelx+1)-1)
    ]

repeatedMat = np.repeat(repeatingMatrix, nele, axis=0).reshape(len(
    repeatingMatrix),nele).transpose()
edofMat = np.add(repeatedMat, repeatedVec)-1
vertedofMat = edofMat.reshape(-1)[1::3].reshape((nele, 8)) #Create
    vertical DoFs array

iK = np.reshape((np.kron(edofMat, np.ones((24,1)))),[24*24*nele,1])
jK = np.reshape((np.kron(edofMat, np.ones((1,24)))),[24*24*nele,1])

jKnew = np.reshape(jK,[1,len(jK)])[0].astype(int)
iKnew = np.reshape(iK,[1,len(iK)])[0].astype(int)

######PREPARE FILTER
iHvalue = int(nele*(2*(np.ceil(rmin)-1)+1)**2)
iH = np.ones((iHvalue*10 , 1))
jH = np.ones(len(iH)).transpose()
sH = np.zeros(len(iH)).transpose()
k = 0
counter = 0

for k1 in range(1,nelz+1):
    for i1 in range(1, nelx+1):
        for j1 in range(1, nely+1):
            e1 = (k1-1)*nelx*nely + (i1-1)*nely+j1
            for k2 in range(int(max(k1-(np.ceil(rmin)-1),1))    ,
     int(min(k1+(np.ceil(rmin)-1),nelz))+1):
                for i2 in range(int(max(i1-(np.ceil(rmin)-1),1))
      ,  int(min(i1+(np.ceil(rmin)-1),nelx))+1):
                    for j2 in range(int(max(j1-(np.ceil(rmin)-1),1)
    )    ,   int(min(j1+(np.ceil(rmin)-1),nely))+1):
                        e2 = (k2-1)*nelx*nely + (i2-1)*nely+j2

                        iH[k] = e1
                        jH[k] = e2
                        sH[k] = max(0,rmin - np.sqrt((i1-i2)**2+(j1
    -j2)**2+(k1-k2)**2))
                        k = k+1

sH = np.reshape(sH,[1,len(sH)])
jH = np.reshape(jH,[1,len(jH)])
iH = np.reshape(iH,[1,len(iH)])

H = sparse.csr_matrix((sH[0], (iH[0]-1 , jH[0]-1)), shape=(nele,
    nele))
Hs = np.sum(H,axis=1)

##START OF THE ITERATION
x = np.tile(volfrac,[nely, nelx, nelz])
loop = 0
change = 1

#Setting values for MMA
xP = x * 1.
```

```python
xformma = x.reshape(-1)
m = 2
xmin = np.zeros((n,1))
xmax = np.ones((n,1))
xval = xformma[np.newaxis].T
xold1 = xval.copy()
xold2 = xval.copy()
low = np.ones((n,1))
upp = np.ones((n,1))
a0 = 1.0
a = np.zeros((m,1))
c = 10000*np.ones((m,1))
d = np.zeros((m,1))
move = 0.2
f = np.zeros((ndof))

#Creating dfdx
selfweightvector = np.zeros(ndof)
selfweightvector[1::3] = 16/ (volfrac * nele) / 8

ce = np.ones(nele)
while change > tolx and loop < 100:
    #start setting up the forces
    x.reshape(n)
    currIndex = 0

    if sw == True:
        for xi in x.reshape(-1):
            totalself =32/ (volfrac * nele)
            selfweight = xi / 8 * totalself #some factor
            if xi>0.5*volfrac:
                currDoF = vertedofMat[currIndex]
                f[currDoF] += selfweight
            currIndex += 1

    ###PRESET F:
    f = f.reshape(-1)

    #Area force on the roof
    #f[1::3*(nely+1)] = 0.01

    #Area force on a certain place
    """
    for i1i in range(0,10):
        for i2i in range(11,32):
            node = np.multiply(i2i, (nelx + 1)*(nely + 1)) + np.
    multiply(i1i, nely + 1)
            f[node*3] = 0.01
    """

    #f[Findex] += 1

    xP = x.reshape(-1) * 1.

    #Setup and solve FE problem
    sK = ((KE.flatten()[np.newaxis]).T*(Emin+(xP)**penal*(E0-Emin))
    ).flatten(order='F')
    K = sparse.coo_matrix((sK,(iKnew,jKnew)),shape=(ndof,ndof)).tocsc()
    freedofsSI = freedofs - 1 #Because of the MATLAB translation

    #FE ANALYSIS
    K = K[freedofsSI,:][:,freedofsSI]
    xguess = U
    resultcgkg = sparse.linalg.cg(K,f[freedofsSI], x0=xguess[freedofsSI],tol=1e-03, M= diags(np.divide(1,K.diagonal())))
    U[freedofsSI]= resultcgkg[0]

    #OBJECTIVE FUNCTION AND SENSITIVITY ANALYSIS
    ce[:] = (np.dot(U[edofMat].reshape(nele,24),KE) * U[edofMat].reshape(nele,24)).sum(1)

    if sw == True:
        ceSelfweight = np.sum(np.squeeze(U[edofMat])* np.squeeze(f[edofMat])* 2,1)
        dcSelfweight = np.sum(np.squeeze(U[edofMat]) * np.squeeze((selfweightvector)[edofMat]) * 2,1)
        dc = (-penal*(E0-Emin)*xP**(penal-1) * ce + dcSelfweight)
        obj = ((Emin+xP**penal*(E0-Emin))*ce + ceSelfweight).sum()
    else:
        obj = ((Emin+xP**penal*(E0-Emin))*ce).sum()
        dc = (-penal*(E0-Emin)*xP**(penal-1) * ce)
    dv = np.ones(nele)

    #FILTERING AND MODIFICATION OF SENSITIVITIES
    dc[:] = np.asarray(H*(dc[np.newaxis].T/Hs))[:,0]
    dv[:] = np.asarray(H*(dv[np.newaxis].T/Hs))[:,0]

    #ROOF CONSTRAINT
    if roofc == True:
        tiledvoid = np.zeros(nele)
        #tiledvoidlist is a list of all the indexes in which a column exist.
        tiledvoidlist = [0,1,2,3]
        tiledvoid[tiledvoidlist]=1

        newvoids = np.around((1-voids))
        total = tiledvoid* newvoids #Total is a matrix with 1's above the void

        xinshape = x.reshape(-1)[total.astype(int)]
        xtotal = np.power(x,5)
        xtotal = np.sum([xtotal[i:i + nely] for i in range(0, len(x), nely)],axis=1)
        epgk = np.exp(10*(1-xtotal)).reshape(-1)
        gcolumns = np.log(np.sum(epgk))/10
        repeateddc = np.repeat(epgk/gcolumns, nely)
        dcrcc = np.zeros((nele))
        dcrcc = np.multiply(repeateddc, total.astype(int))

    xP = xP.reshape(-1)
    x = x.reshape(-1)
    x = np.where(voids, 0.001, x)
    if loop < 5:
        x = np.where(tiledvoid,x*1.1, x)

    ####START OPTIMIZER
    dc = dc.reshape(-1)
    n = nele
    mu0 = 0.001 # Scale factor for objective function
    mu1 = 0.001 # Scale factor for volume constraint function
    fact1 = 0.001

    f0val = mu0*obj
    df0dx = mu0*dc[np.newaxis].T
    volumecons = np.squeeze(mu1*np.array([[x.sum()/n-volfrac]]))
    dxvolume = mu1*(dv/(n*volfrac))[np.newaxis]

    if roofc == True:
        gval = gcolumns * fact1
        fval = np.array([[volumecons],[gval]])
        dxroof = (dcrcc*fact1).reshape(1,n)
        dfdx = np.concatenate((dxvolume,-dxroof*10))

    else:
        fval = volumecons
        dfdx = dxvolume

    xval = x[np.newaxis].T

    xmma,ymma,zmma,lam,xsi,eta,mu,zet,s,low,upp = \
        mmasub(m,n,loop,xval,xmin,xmax,xold1,xold2,f0val,df0dx,fval
    ,dfdx,low,upp,a0,a,c,d,move)

    xold2 = xold1.copy()
    xold1 = xval.copy()
    x = xmma.copy().flatten()

    #Reset F
    f = np.zeros((ndof))
    loop+=1
    # Compute the change by the inf. norm
    change = np.linalg.norm(x.reshape(nelx*nely*nelz,1)-xold1.
    reshape(nelx*nely*nelz,1),np.inf)
    # Write iteration history to screen (req. Python 2.6 or newer)
    print("it.: {0} , obj.: {1:.3f} Vol.: {2:.3f}, ch.: {3:.3f}".
    format(loop,obj,x.sum()/n,change))


fig,ax = plt.subplots()
im = ax.imshow(-x2.T, cmap='gray', interpolation='none', norm=
    colors.Normalize(vmin=-1,vmax=0))

np.savetxt(dirName + "/x.csv", np.around(x,2), delimiter=",")
```
Listing 1: 3D Topology optimization code