Heuristic modification procedures for building spatial design optimization

Snel, T.W.

Award date:
2019

Eindhoven University of Technology

Heuristic Modification Procedures for Building Spatial Design Optimization

Student:
T.W. Snel
0744694

Supervisors:
dr. ir. H. Hofmeyer
ir. S. Boonstra
prof. dr. ir. B. d. Vries

September 23, 2019

M.Sc. Thesis

Project:	Heuristic Modification Procedures for Building Spatial Design Optimization
Status:	Final
Date:	September 23, 2019
Document no.:	A-2019.285
Educational Institute	
University:	Eindhoven University of Technology
Department:	The Built Environment
Master's degree program:	Architecture, Building, & Planning
Research program:	Structural Engineering and Design
Chair:	Applied Mechanics
Graduation Committee	
Chairman:	dr.ir. H. (Hèrm) Hofmeyer
2^{nd} member:	ir. S.(Sjonnie) Boonstra
3^{rd} member:	prof. dr. ir. B. (Bauke) d. Vries
Author	
Student:	T.W. (Tomas) Snel
Address:	Constant Nefkensstraat 5B
ZIP-Code:	5613 ME Eindhoven
Phone no.:	+31(0)613248826
Student no.:	0744694
Email:	t.w.snel@student.tue.nl tomas.snel@gmail.com

Contents

Abstract

This research presents procedures developed for the heuristic optimization of early stage building spatial designs using simulations of co-evolutionary design processes. The research focuses on single disciplinary optimization and it shows that the procedure is capable of optimizing a building spatial design for either structural or building physics performance. Structural performance is defined as the total amount of strain energy, and building physics performance is defined as the thermal energy loss due to heating and cooling of spaces. The constraints for optimization are equal volume, equal space count, no overlap between spaces, and no floating spaces. The procedure is divided into three sub procedures: performance evaluation, space ranking, and building modification. For all stages multiple parameters are developed and elaborated. For the performance evaluation procedure it is shown that structural evaluation should rate spaces from best to worst as highest to lowest strain energy. Building physics evaluation should rate spaces from best to worst as lowest to highest thermal energy loss. Furthermore it can be stated that combining a geometrical clustering approach with the evaluated performances is capable of improving a building spatial design for both disciplines. The space ranking in this study is straightforward since only one discipline is considered at a time. For the building modification sub procedure the most effective parameters are those that move a building spatial design towards the most optimal shape for the given discipline. For structural optimization this means a one storey high, square floor plan. Building physics optimization finds the best solutions when the modifications move the building spatial design towards a cuboid design. Unfortunately, a part of the found solutions result in undesirable building spatial designs. These designs contain spaces that have unrealistic ratios between their height, width and depth.

Keywords: Heuristic optimization, Structural optimization, Building physics optimization, Early stage building spatial designs.

1 Introduction

This graduation project is inspired by the project *"Excellent building via forefront MDO. Lowest energy use, optimal spatial and structural performance"*, which is funded by the Dutch foundation for technology sciences: NWO-TTW (grant number 13596). The NWO-TTW project aims to develop and research techniques for multidisciplinary optimization (MDO) of building spatial designs in the Architecture, Engineering, and Construction (AEC) industry. Two disciplines are selected for optimization; namely structural design (SD) and building physics (BP). The objectives for these disciplines are minimal strain energy (SD), and minimal heating/cooling energy (BP). Three methods have been developed for the optimization (a) an Evolutionary Algorithm (EA) called SMS-EMOA, (b) a Simulation of Co-evolutionary Design Process (SCDP) and (c) a combination of EA and SCDP. Method (a) uses a mathematical approach in combination with a constricted design space in order to find high quality Pareto front approximations (PFA). While a PFA can be found within the given design space, the computational cost is high and the global optimum might not be included in the design space. The second method (b) applies a heuristic approach, to implement engineering knowledge as optimization rules. This method allows for an exploration of the design search space, possibly altering it in each cycle of optimization. Although optima are not found, this method generally can find better designs in one or more disciplines. Combining these methods into (c) can provide an approach that is capable of finding high quality PFAs with altering design search spaces.

The research and development of the before mentioned heuristic rules are the topic for this graduation project. Hofmeyer and Davila Delgado (2015) show that heuristics can greatly reduce the amount of computational time required in building spatial optimization (BSO) for structural optimization. They optimize a building spatial design by simply removing the 50% worst performing spaces, then scaling the remaining spaces over the x and y axis, and split the remaining spaces once across their longest dimension. The procedure of this algorithm, as illustrated in Figure 1.1, and the Building Spatial Design Toolbox (BSO Toolbox) as developed by (Boonstra et al., 2018) are taken as starting point of this graduation project.

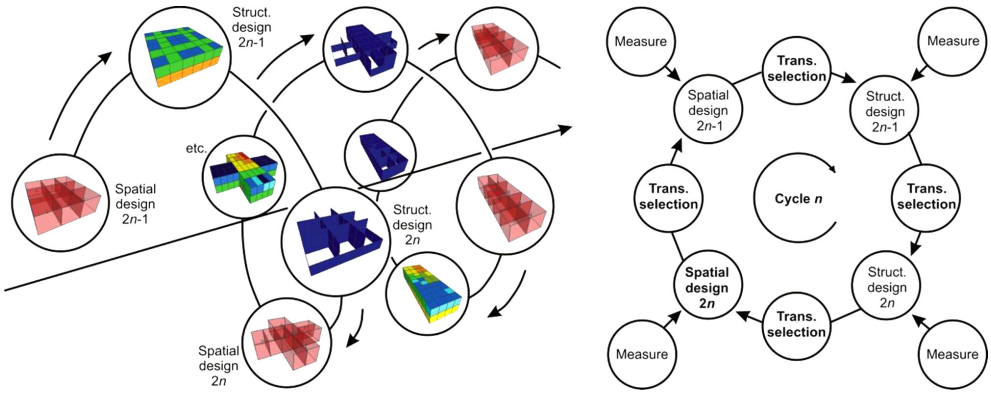


Figure 1.1: Procedure of the SCDP as developed by Hofmeyer and Davila Delgado (2013)

1.1 Research Goals

The research goal of this graduation project is to *develop new heuristic rules to optimize early stage building spatial designs following the SCDP procedure*. The developed SCDP procedure will be applied on two disciplines currently implemented in the BSO Toolbox: SD and BP. While the development of heuristic rules does include procedures that enable multidisciplinary simulations, results focuses only on single disciplinary studies. In order to achieve the goal two research questions have been formulated:

- How to interpret performance evaluations of an early stage building spatial design?
- How to use that interpretation to modify a building spatial design such that its performance improves?

1.2 Report Overview

Some general definitions and concepts are introduced first in Chapter 2, which covers the need for optimization procedures in early stage building design processes and the basic formulas of optimization. Chapter 3 describes the formulations and representations of the BSO Toolbox which is used for this graduation. In Chapter 4 the developed heuristic procedure are explained. Results of two parameter studies on single disciplinary simulations with the new procedures are presented and analyzed in Chapter 5. After which a critical discussion and the conclusions are presented in Chapters 6 and 7 respectively. Finally recommendations are presented in Chapter 8.

2 Need For Optimization

The AEC industry is responsible for 40% of the annual usage of energy and resources worldwide (European Construction Tehnology Platform, 2005) and with the climate agreement of Paris 2015, the goal is set to reduce global emissions. In order to achieve more optimal building designs, and thus reducing energy and resources, engineering simulations are becoming the norm for practice in the AEC industry. However, the potential for this technology to support early-stage building design decisions is not fully utilized, because current tools and processes do not support rapid generation and evaluation of building design alternatives (Flager & Haymaker, 2007).

2.1 Designing Buildings

Horváth (2005) defines a design process as '*an iterative search process in which designers gather, generate, represent, transform, manipulate, and communicate information and knowledge related to various domains of design concepts*'. So designers iterate over multiple design options and alternatives to find a design that meets all performance criteria to a satisfactory level. A correlation is found between the amount of alternatives, or iterations, and the quality of the final chosen design (Akin, 2001), where a higher amount of alternatives correlates with a higher quality final design. Unfortunately, on average only three alternatives are considered in a normal design process before the final design is chosen. Experts in the AEC field spend on average over a month to generate and evaluate a single design alternative (Flager, Welle, Bansal, Soremekun, & Haymaker, 2009). The inability to quickly generate and analyze multiple alternatives considering multiple disciplines leaves a large area of the design search space unexplored. Flager and Haymaker (2007) state that, if engineers are involved in the early stages of a building design they are mainly validating alternatives rather than generating the design alternatives. Nevertheless, all decisions that have been made early in the design process have a significant impact on the life-cycle, economic, and environmental performances of buildings. Okudan and Tauhid (2008) find that an ill-defined conceptual design is almost never fixed at a later stage. Additionally Rezaee, Brown, Haymaker, and Augenbroe (2018) state that there is no practical framework for designers to generate more promising alternatives regarding energy performance in the early design stage phase. The software used to analyse and optimize various parameters in building spatial designs are generally based on detailed building simulations (Machairas, Tsangrassoulis, & Axarli, 2014). Such simulations are not suitable for conceptual designs, because they are computationally expensive and require many assumptions for design decisions that are decided in a later design stage.

The complexity of design processes in the AEC industry is another reason why optimization is rarely applied in the exploration of design search spaces. Each discipline by itself is complex, but an additional layer of complexity is added if two or more disciplines would be considered simultaneously. Often disciplines contradict each other and no one solution can be found that is optimal for all. Figure 2.1 shows a building spatial design being optimized towards either SD or BP. However the optimal results differ from one another so it is unclear what the optimal result is. Besides contradiction, an interdependency of parameters exists as described by Haymaker, Kunz, Suter, and Fischer (2004) and Ritter, Geyer, and Borrmann (2013). Research has been conducted in all disciplines of the AEC industry. Many of those are optimizations of a specific parameter or discipline, but rarely focused on holistic searches that cover an array of disciplines and objectives.

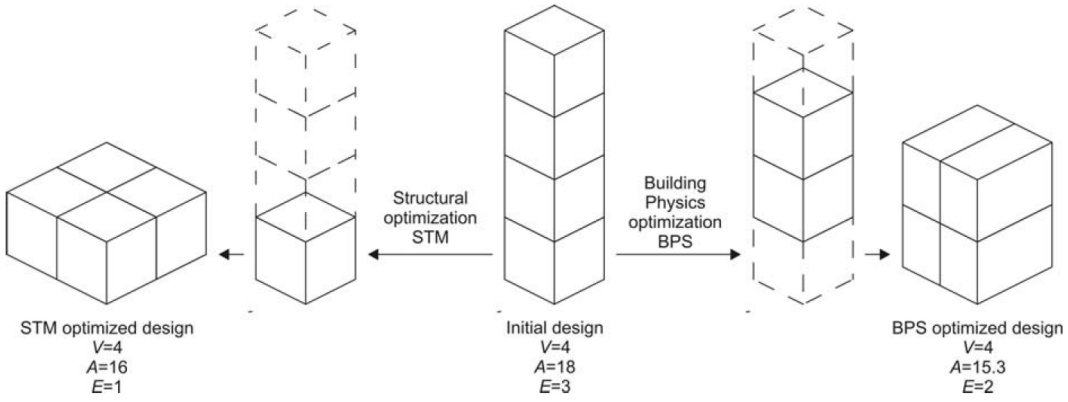


Figure 2.1: A building spatial design that has been optimized twice, once for structural and once for building physics objectives. Where the SD optimal design goes towards minimal building height, the BP design moves to minimal surface area resulting in two different solutions. (Hofmeyer & Emmerich, 2013)

2.2 MDO in the AEC Industry

In many industries multidisciplinary optimization is common practice when dealing with complex design decisions. However, such tools do not exist to support the AEC industry in early stage building design development. Incorporating performance models into existing software to guide the generation process will yield tools that can help architects, designers, and engineers to think critically yet qualitatively from the viewpoint of different disciplines. Unfortunately, providing these tools for specific design conditions and performances remains a challenge (Shea, Aish, & Gourtovaia, 2005). Many of the current analysis and simulation tools focus on a single discipline. However, it is necessary to consider disciplines simultaneously. The following section will elaborate on single- and multi-objective optimization.

2.2.1 Optimization in general

Optimization can be defined as *'the procedure of finding the minimum or the maximum value of a function by choosing a number of variables subject to a number of constraints'* (Machairas et al., 2014). This function is commonly known as the objective function. Equation 2.1 gives the mathematical form of a generic single-objective optimization problem. X is the design search space that includes all possible designs x where *f(x)* is only evaluated if the equality and inequality constraints $g(x)$ and $h(x)$ hold.

$$\begin{aligned} &\text{minimize } f(x) && x \in X \\ &\text{subject to } g_j(x) \leq 0 && j \in \{1, 2, ..., m\} \\ &\qquad h_k(x) = 0 && k \in \{1, 2, ..., n\} \end{aligned} \tag{2.1}$$

Different techniques have been developed to solve these single-objective optimization problems, e.g. expert systems, gradient optimization, evolutionary algorithms, and design grammars (Hofmeyer & Davila Delgado, 2015). As mentioned in section 2, much of the research in the AEC industry focuses on only a single discipline. Examples of optimization in the AEC industry are: Energy consumption by Cheung, Fuller, and Luther

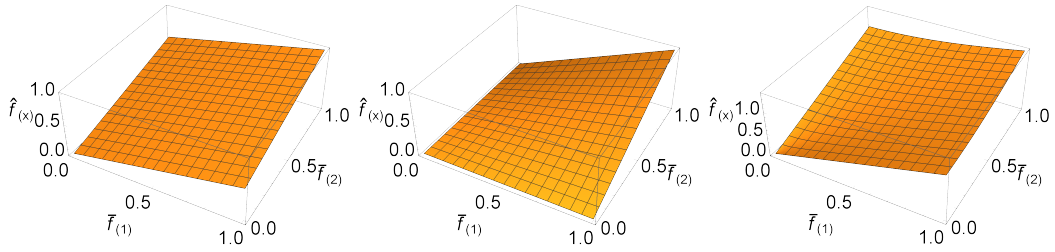
(2005), and Friess, Rakhshan, Hendawi, and Tajerzadeh (2012). User comfort is optimized by Bouchlaghem and Letherman (1990), and Torres and Sakamoto (2007). An absorption chiller system is optimized by Chow, Zhang, Lin, and Song (2002), and an HVAC envelope design by Bichiou and Krarti (2011). Structural stiffness is optimized by Fuyama, Law, and Krawinkler (1997), and buckling load of a stayed columns by Verbeeck, Van Steirteghem, De Wilde, and Samyn (2005)

2.2.2 Multi Objective Optimization

Translating a single-objective to a multi-objective optimization can be achieved by extending equation 2.1 into equation 2.2. The extension allows for new objectives to be evaluated for solutions in the design search space. Often no one single solution exists that is optimal for all objectives, a trade-off situation occurs when one objective cannot be improved without declining others. Two major approaches can be identified to express optimality of multi-objective problems: aggregation and Pareto optimality.

$$\begin{aligned} &\text{minimise } f_i(x) && i \in \{1,2,...,l\}; x \in X \\ &\text{subject to } g_j(x) \leq 0 && j \in \{1,2,...,m\} \\ &h_k(x) = 0 && k \in \{1,2,...,n\} \end{aligned} \tag{2.2}$$

Aggregation Functions The classic solution to deal with multi-objective problems is to rewrite them into a single-objective problem. Using aggregation functions objectives can be combined, normally by normalizing objectives, and summing or multiplying them into one objective to assess the problem as if it were a single-objective problem. Figure 2.2 shows different aggregation functions and their resulting values, the x and y axis describe the (normalized) objectives $\overline{f_1}(x)$ and $\overline{f_2}(x)$ the z axis gives the aggregated result, $\hat{f}(x)$.



(a) Sum *(b) Product* *(c) Euclidean distance*

Figure 2.2: Results from aggregation algorithms. The x and y axis show the normalized performance $\overline{f_1}(x)$ and $\overline{f_2}(x)$, the z axis shows the aggregated result $\hat{f}(x)$.

Pareto Frontiers The second approach in MDO is inspired by Pareto (1896): a so called Pareto Front, see figure 2.3. A Pareto Front is defined as the set of Pareto optimal solutions, where a solution is Pareto optimal when for each of the objectives no objective can be improved without diminishing another objective. During an optimization process a Pareto Front Approximation (PFA) can be constructed. To achieve the PFA each solution is evaluated whether it is Pareto optimal for the solutions found at that time. The PFA is the set of Pareto optimal solutions until every solutions in the design search space has been evaluated, then it becomes the Pareto Front.

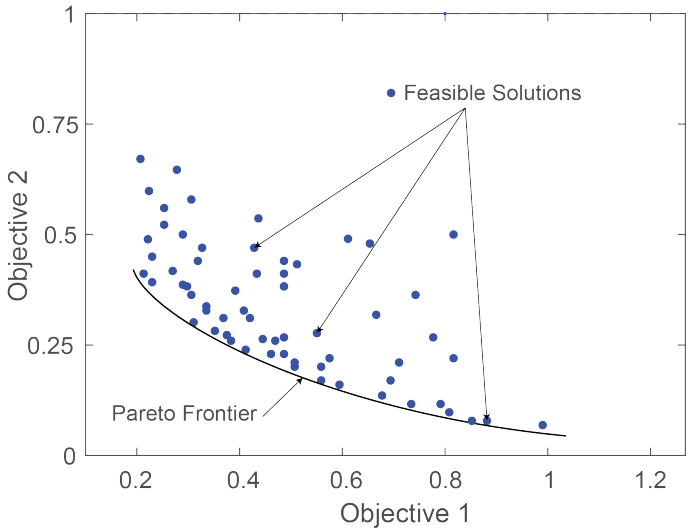


Figure 2.3: An example of a Pareto Frontier

Where which algorithm is used depends on the given problem; e.g. for some problems the analytical solution can be solved. Adamski (2007) and Marks (1997) mathematically describe the shape of a building and solve it with a numerical solution, finding the true optimum. Jedrzejuk and Marks (2002) optimize the shape, functional structure and heat utilization of single and multi-family houses using an analytical approach. In other cases a Pareto based solution is required, e.g. D'Cruz and Radford (1987) optimize a simple building with the objectives of thermal load, daylight, planning efficiency and capital cost. Flager et al. (2009), who use a PFA in their multidisciplinary optimization of a classroom.

2.3 Cluster Approaches

Data clustering is the process of grouping together (multi-)dimensional data-points into a number of clusters or bins (der Merwe & Engelbrecht, 2003). Clustering is used to gain a better understanding of data sets and therefore it is applied to a wide variety of problems, e.g. exploratory data analysis, data mining, and mathematical programming. An operational definition is given by Jain (2010) as *'Given a representation of n-objects, find K-groups based on a measure of similarity such that the similarities between objects in the same group are high while the similarities between objects in different groups are low'.* The similarities that determines clusters will alter for each problem and is based on the definition of the objects. Clustering algorithms are divided into two classes: supervised and unsupervised. If cluster parameters are predefined and an external source states to which cluster a data-point belongs it is called supervised. However, in many cases the potential clusters are not known a priori. For these unknown situations unsupervised algorithms are developed which create clusters based on the distance of one data-point to another.

The most well-known unsupervised cluster algorithm is K-means, Steinhaus (1956), Lloyd (1982), Ball and Hall (1965), and MacQueen et al. (1967). It is defined as follows: *Let $X = x_i, i = 1, ..., n$ be the set of n d-dimensional points to be clustered into a set of K clusters, $C = c_k, k = 1, ..., K$. K-means algorithm find a partition such that the squared error between the empirical mean of a cluster and the points in the cluster in minimized* (Jain, 2010).

3 Building Spatial Optimization Toolbox

This section describes the Building Spatial Optimization toolbox (BSO Toolbox) as presented by Boonstra et al. (2018). Figure 3.1 shows the organisation of the BSO Toolbox. The different elements in the BSO Toolbox and their interaction will be explained and elaborated on. First it should be stressed that the functions and representation described here are to evaluate building spatial designs and not to optimize them. The optimization algorithms will be discussed in the Chapter 4.

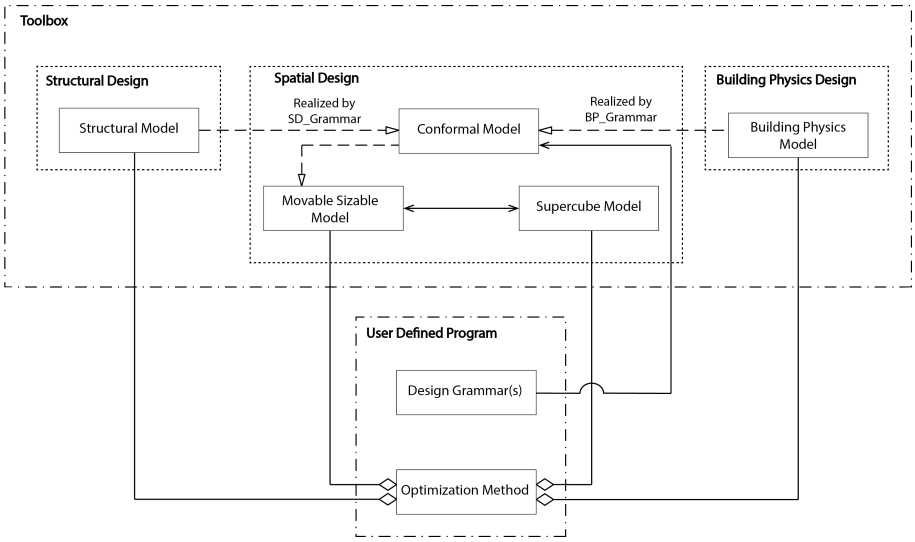


Figure 3.1: Organisation of the BSO Toolbox as developed by (Boonstra et al., 2018)

Representations Three different building spatial design representations are implemented in the BSO Toolbox. The first two are building spatial representations: Supercube (SC) and Movable-Sizable (MS), Figure 3.1. The third available representation is a conformal model. The latter representation helps to translate a building spatial design model into domain specific models. In this graduation project the MS-representation is elaborated in the following paragraph. For the other representations the reader is reffered to Boonstra et al. (2018).

The MS-representation uses a single vector $\boldsymbol{S}$ to collect all spaces S_i. Expression 3.1 shows the formulation and figure 3.2 shows an example of the representation. Each space s_i has two sets, the set C describes the origin and set D defines the dimensions (width, height, depth) of the space. Adding spaces can be done by simply appending its properties to the $\boldsymbol{S}$ list. The MS-representation is a super structure free approach which allows for spaces to be freely modified, deleted and created during optimization. Modification of the building spatial design is achieved by altering, adding or removing elements of the $\boldsymbol{S}$ list. However, since spaces are not interconnected, modifications may lead to infeasible designs. When a space is modified it may overlap with other spaces, this obviously leads to infeasible building spatial designs. Another limiting property is the formulation of the MS-representation which only allows orthogonal building spatial designs to be formulated. This alters some of the ideal shapes that can be constructed in the design search space, e.g. a cubical shape is now the ideal ratio for surface to volume instead of a spherical shape.

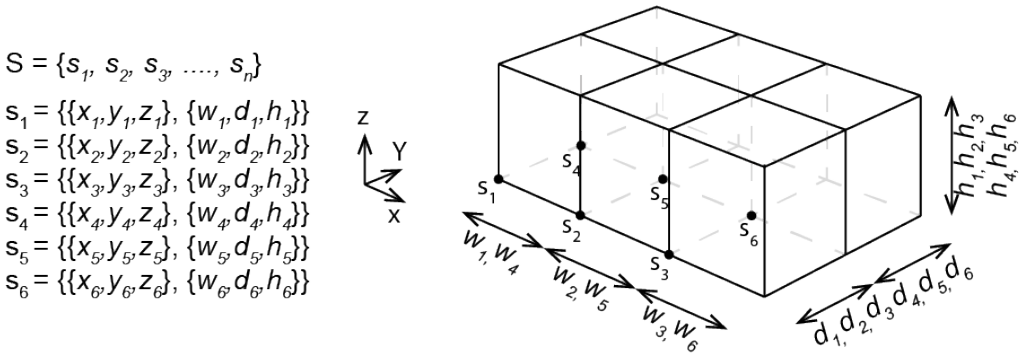


Figure 3.2: Movable-Sizable building representation

$$\boldsymbol{S} = \{S_1, S_2 \ldots, S_{N_{spaces}}\}$$
where
$$S_i = \{C, D\} \qquad C = \{x, y, z\} \qquad D = \{w, d, h\} \tag{3.1}$$

Building Analysis The BSO Toolbox uses the Finite Element Method (FEM) as a computational analysis for structural performance. After the calculation has been finished the structural strain energy (in Nmm) can be retrieved per space, or for the entire building. For each space a performance is computed as the sum of strain energy in the walls and floor that construct the space.

Thermal performances of a building spatial designs are measured by simulating the amount of energy that is required in each space to keep the space's temperature between a lower and upper limit. Thermal energy loss is calculated by means of a Resistor-Capacitor network representation (RC-network) (Boonstra, 2016), which only requires a low level of detail of the building making it suitable for the used representations. The BSO Toolbox provides the total energy loss for both the entire building or a single space (in kWh).

3.1 User Defined Program

A second structure, as illustrated in Figure 3.1, is required to provide input and request output: the user defined program (UDP). The UDP consists of two parts: optimization method(s) and grammars. The optimization method provides the BSO Toolbox with building spatial design models, and uses the results of the BSO Toolbox analysis to optimize the given building spatial design. To use a grammar, the user defines settings, which are then used by the grammar to generate a domain specific model.

3.1.1 Grammars

In order to simulate and calculate the building spatial designs put into the BSO Toolbox, a transformation from spatial representation to discipline specific model is required. Each discipline in the BSO Toolbox requires its own grammar(s) with the discipline specific settings. The structural grammar includes elements such as (structural) material properties, loads, and the amount of divisions to be made in the Finite Element analysis. The BP grammar adds environmental data, an amount of time steps per hour that need to be considered and thermal properties of building materials.

Structural Grammar The grammer and settings that are used for the structural discipline in this graduation project assign flat shell components to each wall and floor in the building spatial design. These flat shell components have a thickness of 150 mm and material properties that resemble concrete, i.e. a Youngs modulus of 30000 N/mm^2 and a Poisson's ratio of 0.3. Five load cases have been added: a life load with a magnitude of 5 kN/m^2 in -z direction on each horizontal flat shell component. Four wind load cases, +x, +y, -x, and -y, on each outside surface. The wind load itself has three components, pressure 1.0 kN/m^2, suction 0.8 kN/m^2, and shear 0.4 kN/m^2. Finally line supports are added at the edge of each space with the coordinate $z \leqslant 0$. The FEM model is meshed using three divisions in each direction.

Building Physics Grammar Also for the building physics grammar material properties are applied that resemble concrete are used: a specific weight of 2400 kg/m^3, specific heat capacity of 850 $J/(K \times kg)$, and a thermal conduction coefficient of 1.8 $W/(K \times m)$. The parts of walls and floors which are only part of one space are assigned an additional layer to their structure, namely a layer of insulation with a thickness of 150 mm with a specific weight of 60 kg/m^3, specific heat capacity of 850 $J/(K \times kg)$ and a thermal conduction coefficient of 0.04 $W/(K \times m)$. The temperature set points are 20 ℃ for heating and 25 ℃ for cooling, the total available heating and cooling in spaces is set to 100 W/m^3. The ventilation rate for each space in the design is one air change per hour. Real world data which was measured in De Bilt, The Netherlands, by the Dutch Royal Meteorological Institute (KNMI) is used for the temperature profile of the weather and the ground profile has a constant temperature of 10 ℃. The simulations use two periods of three days each, a typical warm period (July 2nd - 4th 1976) and a typical cold period (December 30th 1978– January 1st 1979). Before these periods, a warm up period of four days is added in order to start the periods with appropriate initial temperatures.

3.2 Optimization approach

The initial heuristic optimization approach is defined in the article published by Hofmeyer and Davila Delgado (2015). This single-objective approach rank the spaces from high to low strain energy and removes the 50% of the spaces with the lowest strain energy, scaled the remaining spaces over the x- and y-axis to regain the initial volume and divided all remaining spaces once over their largest span. This graduation project focuses on new heuristics for the UDP.

Constraints In order to focus on feasible designs 2 constraints are defined: overlapping and floating spaces. The possibility of overlapping spaces is enabled due to the definition of spaces in the MS-representation, where each space is defined individually without any direct links to other spaces. Overlap between spaces occurs due to modification techniques, the procedures in the HBO toolbox are developed in such a way that overlap of spaces does not occur. Floating spaces can occur when a space loses all its surrounding spaces due to removal procedures. For these spaces a procedure is developed that checks for each space if it is floating. If a design contains one or more floating spaces the cycle is considered to the final one and the simulation is terminated. Another two constraints are defined in order to keep comparable building spatial design throughout simulations: equal volume and space count, i.e. the volume and the amount of spaces of the new building spatial design should be equal to the initial building spatial design. Of course when a building spatial design is made smaller or lower the performances for structural design

and building physics will increase. However, this is not part of the design brief provided by the initial building spatial design. A snap on function is implemented which prevents numbers with a large amount of decimals to occur. This function might alter the volume slightly. However if the difference in volume is within 1% this is considered to be satisfying for the volume constraint.

4 Heuristic Building Optimization Algorithm

The procedure of the heuristic building optimization (HBO) method can be divided into three separate sub procedures namely: performance evaluation, space ranking, and building modification. Each of these sub procedures offers several options, applying these options in a combinatorial fashion allows a single building spatial design to be modified into a large variety of new designs which results in different design modifications. First, the separate procedures are elaborated, after which an example cycle is given. The exact procedure and codes developed for this research can be found in appendices A and C.

4.1 Normalized Performances

The HBO procedure is built for normalized performance values of spaces due to the different nature of the included disciplines. The calculated performances for each discipline is built up of different type of units, which cannot be compared one to one due to difference of magnitude or being a complete different kind of unit. The normalization process is described in Equations 4.1a and 4.1b. When normalized, the range of all values is between 0 and 1, with 1 being the best possible performance. The difference between the equations can be found in which performances rate highest. Where 4.1a rates a high performance value as best, whereas 4.1b rates a low performance value as best. Which equation is used can be specified for each individual discipline.

$$F_{x,norm} = \frac{F_x - F_{x,min}}{F_{x,max} - F_{xmin}} \tag{4.1a}$$

$$F_{y,norm} = \frac{F_{y,max} - F_y}{F_{y,max} - F_{y,min}} \tag{4.1b}$$

4.2 Performance Evaluation

In the Performance Evaluation procedure the results provided by the BSO toolbox are evaluated. Firstly it must be determined whether the spaces in a design are considered individually or as clusters of spaces. Arbitrary decisions can occur when identical spaces need to be ranked, these arbitrary decisions can be parried when using clusters to assess a building spatial design. Unfortunately, assessing clusters can lead to difficulties and restrictions in the building modification stage while the individual selection allows for more precise modifications, i.e. if the cluster of removed spaces differs in size with the one of spaces that are split.

Two different cluster approaches are available. The first one is the K-means algorithm, as described in section 2.3, which clusters the spaces based on their performances. The second type is based on the geometrical properties of spaces. It clusters six times, for all six different facades in a building. I.e. the north, east, south, west facade and the upper and lower floor, see figure 4.1. The procedure searches the minimum and maximum $x, y,$ and$z$, for each of those values it searches the space with the largest width, depth and height respectively. The six clusters are then constructed using these values as boundaries and adding each space within to its cluster. Lastly, a set of spaces is created, which consists out of the spaces that were not clustered in any of the other clusters. The performance of such a geometrical cluster is set as the average value of the performances of the individual spaces in that cluster.

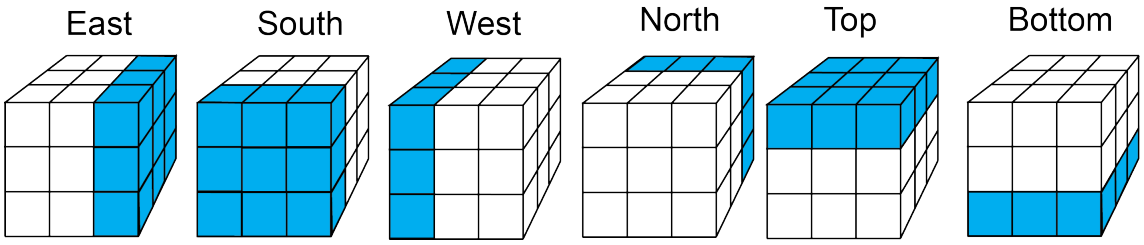


Figure 4.1: Six different geometrical clusters, a seventh set is constructed with all spaces that are not included in any of the highlighted clusters.

After a choice has been made for the individual or clustered approach the provided performances are processed. Note that this can be any number of disciplines. When only one discipline is required, the evaluation is straightforward for the provided performances. For a SCDP with multiple disciplines aggregation and non-aggregation of procedures are available. Aggregation methods implemented are: summation, product, and euclidean distance functions. When performances are non-aggregated they can still be altered with weight functions, allowing the steering of solutions in the design search space.

4.3 Space Ranking

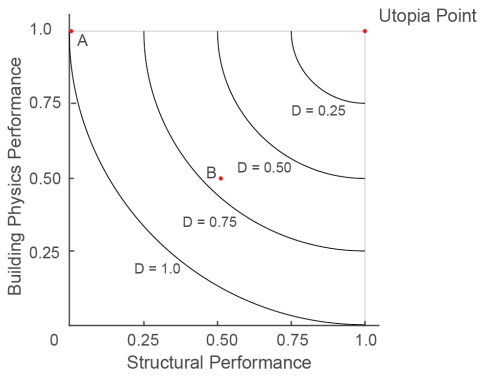
Spaces in the building spatial design are ranked based on their performance evaluation. If only one performance per space is given such a ranking is straightforward the highest to lowest value listed. However, when a non-aggregated procedure is used it is not straightforward which space is best or worst. Six different options are implemented to achieve possibly different rankings. The first ranking is made on the criteria of best performance, where best is taken as the shortest distance to the utopia point (the best possible point). An opposite criteria can be set as well, ranking based on dystopian point (the worst possible point). While in a one dimensional list both criteria will result in identical lists, when in a multidimensional environment the list differs. E.g. a design which has two points namely A (1, 0) and B (0.5 ; 0.5). For the best performance criteria point B has the shortest distance towards the utopia point and is thus considered better than point A. However, for the worst performance criteria point B is again the space with the shortest distance, however now towards the dystopia point, see Figure 4.2. The third and fourth approach are ranking while altering between disciplines to select either the best or worst performing space.

4.4 Building Modification

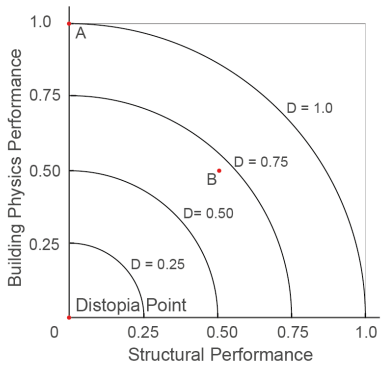
The last part of the HBO procedure is modifying the building spatial design using the data constructed in the other sub procedures. This part contains three steps: space removal, enforcing volume constraint, and enforcing space count constraint.

Space Removal The first step is to remove an amount of spaces based on the ranking generated in the previous stage. For this six functions are implemented that differ in the amount of spaces they remove. The options implemented are: one space/cluster, 10%, 20%, 30%, 40%, or 50% of all spaces removed.

Enforcing Volume Constraint After spaces are removed the new design has to meet the volume constrain, which is set to the initial design. To comply to this volume constraint



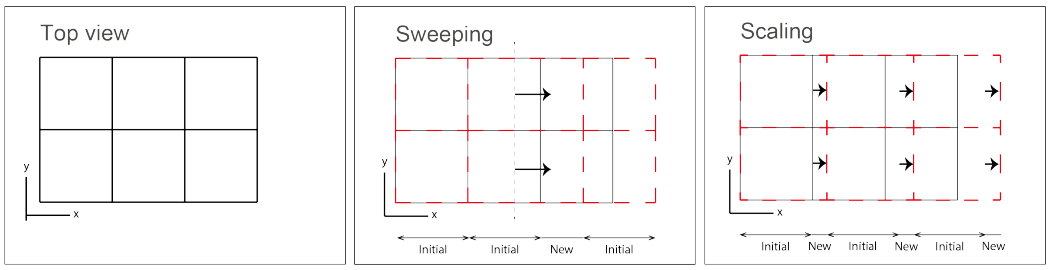
(a) Utopia point with distances towards it



(b) Distopia point with distances towards it

Figure 4.2: Utopia and Dystopia point with projected distances towards the points

two types of functions are implemented: scaling and sweeping, as illustrated in Figure 4.3. The scaling functions multiply the coordinates of spaces with the ratio between initial and new design volume up until the volume constraint is met. Scaling allows for modification over all three axis, and all combinations of them. Sweeping selects spaces on a given orthogonal plane and extrudes all those spaces until the volume constrain is met. The effectiveness of each modification option is dependent on the building spatial design type, e.g. a highrise building will not improve when scaling or sweeping upwards in height but a lowrise building might improve. Both the scaling and sweeping procedure preclude the possibility of overlap between spaces.



(a) Top view of a 3x2 building spatial design

(b) Top view, in red the swept building spatial design

(c) Top view, in red the scaled building spatial design

Figure 4.3: Building Modifications scaling and sweeping on a 2x3 building spatial design

Enforcing Space Count Constraint The last part of the modification procedure is to satisfy the space count constraint, this can be achieved by splitting specific spaces in the new design into two. The four options for splitting are to split the largest, the smallest, the best, or the worst space. For a designer it might feel normal to split the largest space since it has the most potential to split into two feasible spaces, something that can be an issue when splitting the smallest space. That can result into two spaces that are too small to be used, however it gives opportunity to smoothly evolve a building design over multiple iterations. Splitting the space with the best performance creates more spaces around coordinates with good performance results. Splitting the worst performing space altering its properties provides opportunity for it to improve its performance or for it to gradually be removed in multiple cycles. A space will always be split over its largest

dimension. In case of two or three equal largest dimensions a preference can be set across which dimension to split first. Dimensions smaller than $2000mm$ cannot be split, since this would result in unrealistically small spaces.

4.4.1 Infeasible building spatial designs

It is possible that infeasible building spatial designs occur. The procedure to detect floating spaces is as follows: First a list is made with all spaces that have a z coordinate where $z \leq 0$, i.e. they are not floating. Then for each wall or floor of the non floating spaces is checked whether another space is connected to it, if a space is connected that space is also marked as not floating. This cycle continues until no more spaces can be added to the non floating list. Then it is checked whether all spaces are marked as not floating, if so the building design is considered feasible. If a building spatial design is considered infeasible the HBO procedure will not allow the SCDP to enter into a new cycle.

4.5 A possible cycle

The sub procedures are written such that they chronologically follow each other in a SCDP. Figure 4.4 illustrates a SCDP, the procedures from an initial building spatial design up to the new building spatial design is considered a single cycle, at the new building spatial design a new cycle starts. The first step is to calculate the discipline performances of a building design, these performances are then normalized. Then spaces are clustered by using the K-means procedure. After which for each cluster the performances are squared. Since all clusters now have one performance value, they can easily be ranked. From this ranking the cluster with the lowest performance is selected and all spaces in that cluster are removed from the building spatial design. The remaining spaces are scaled over the x and y axis by the square root of the initial volume divided by the current volume. Finally the largest space in the building spatial design is split across its largest dimension, this repeats until the space count constraint is satisfied.

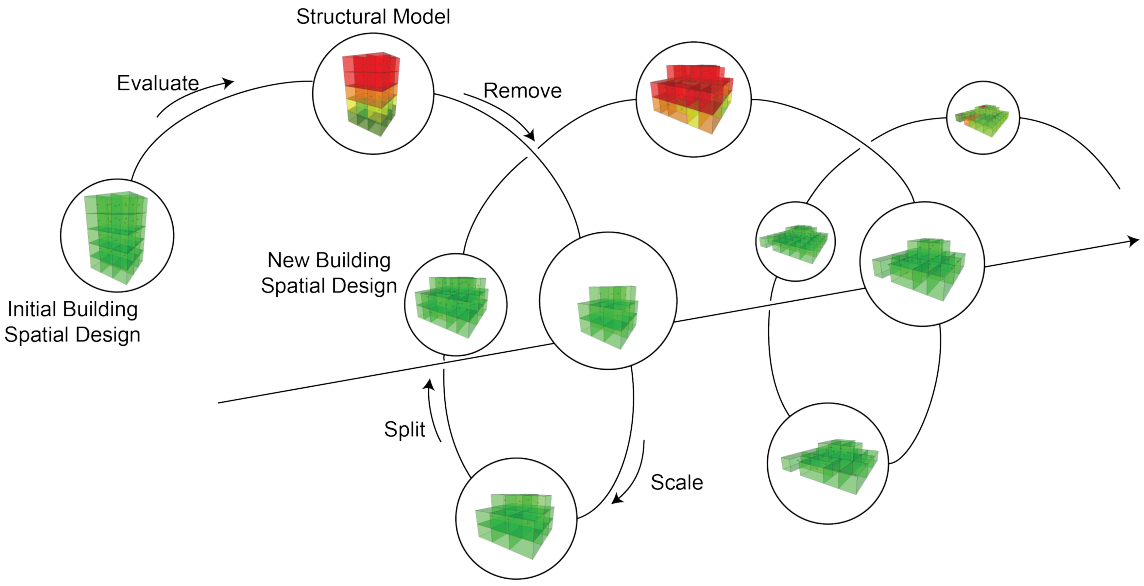


Figure 4.4: A simulation of co-evolutionary design process for structural optimization. From initial building spatial design up to the new building spatial design is considered as a cycle. A SCDP can contain multiple cycles

5 Results

Two parameter studies are carried out for this research. Their results are described in the following sections, the Figures belonging to the results are presented in appendix B. Simulations with parameter set one are presented first. Using the results and conclusions of set one the procedures are adjusted for parameter set two. Due to the complexity and vastness of possible parameter combinations, a selection is made which parameters to research and elaborate. Simulations and analyses in this graduation project only include single-disciplinary optimizations. The governing result of each simulation is the cycle that has the best performing building spatial design, no matter on which cycle this result is found.

Three different building spatial designs are optimized, see Figure 5.1. All spaces of the initial designs have a width, depth, and height of three meters. The first design is a cubic building, with a three times three spaces floor plan and three storeys in height. The second design is a lowrise building consisting out of a three times eight floor plan with three storeys. The third design is a highrise building constructed with a three times three floor plan stretching out over nine storeys in total.

The results of these parameter studies are presented using boxplots, where the top and bottom are the 25th and 75th percentiles of the data. The distance between the top and bottom of the box is defined as the interquartile range. The line in the middle of the box is the median of the data. Whiskers are drawn from the ends of the box to the furthest data point within whisker length. The maximum length is defined as 1.5 times the interquartile range, all data points outside that are considered outliers and marked with a dot sign. The diamond indicates the average value of the data set.

5.1 Simulations Using Parameter Set 1

Set one only considers individual assessment of spaces and includes the parameters as shown in table 5.1.

	Parameters
Performance Assessment	Individual
Best Performance	SD: High or Low, BP: Low
Removal Options	1 space, 10%, 20%, 30%, 40%, 50 %
Number of cycles	So that the amount of spaces removed over all cycles equals at least the space count of the design
Scaling Options	X, Y, Z, XY, XZ, YZ, XYZ
Sweeping Options	X, Y, Z axis at begin, middle or end
Splitting Options	Largest, Smallest, Best, or Worst space
Splitting Preference	If the height is more than 6000mm: Z-X-Y, else X-Y

Table 5.1: The parameters used in the first simulation set

5.1.1 Structural Optimization

Performance Normalization The HBO procedure allows for simulations with either preference for high or low strain energy. Previous research (Hofmeyer & Davila Delgado, 2013) assumed that a high strain energy in the structural volume is beneficial for the structural system, i.e. the volume is highly utilized. However, no research was presented to confirm this assumption. The HBO approach allows for simulations with either preference to high or low strain energy. Figure 5.2 shows the difference between considering high

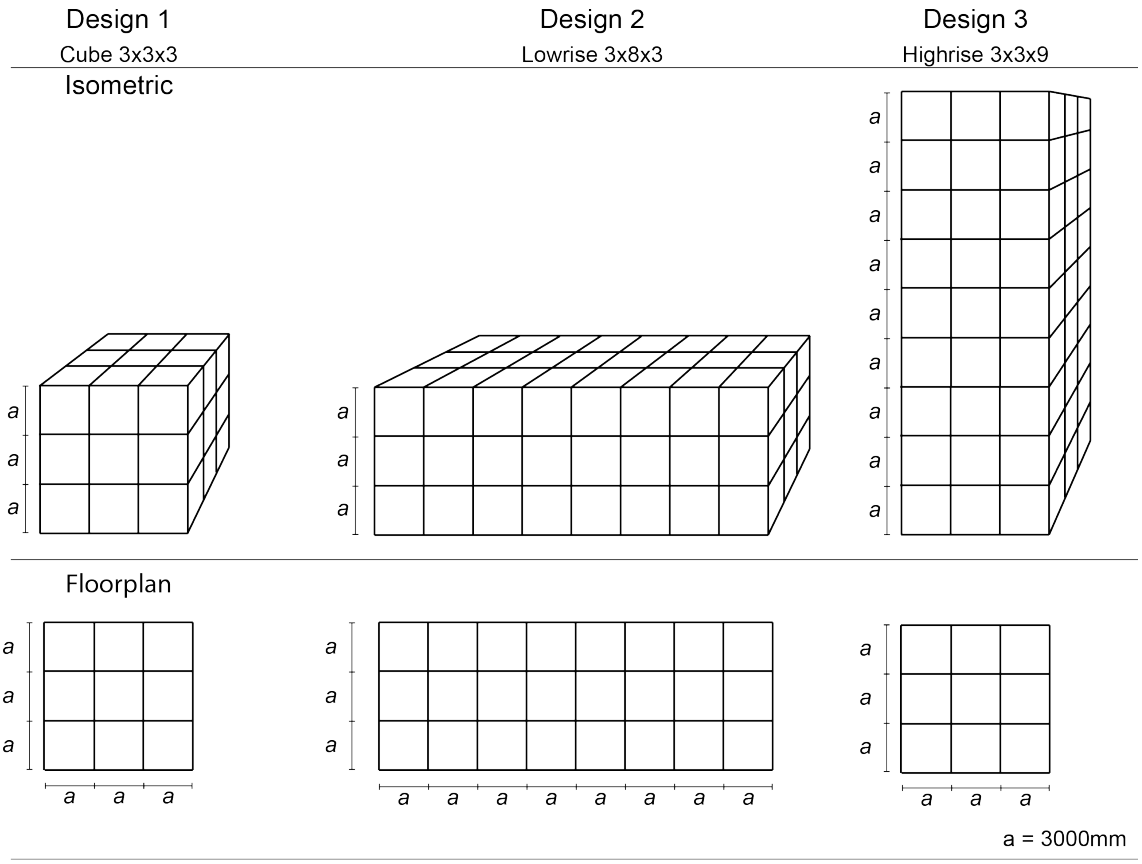
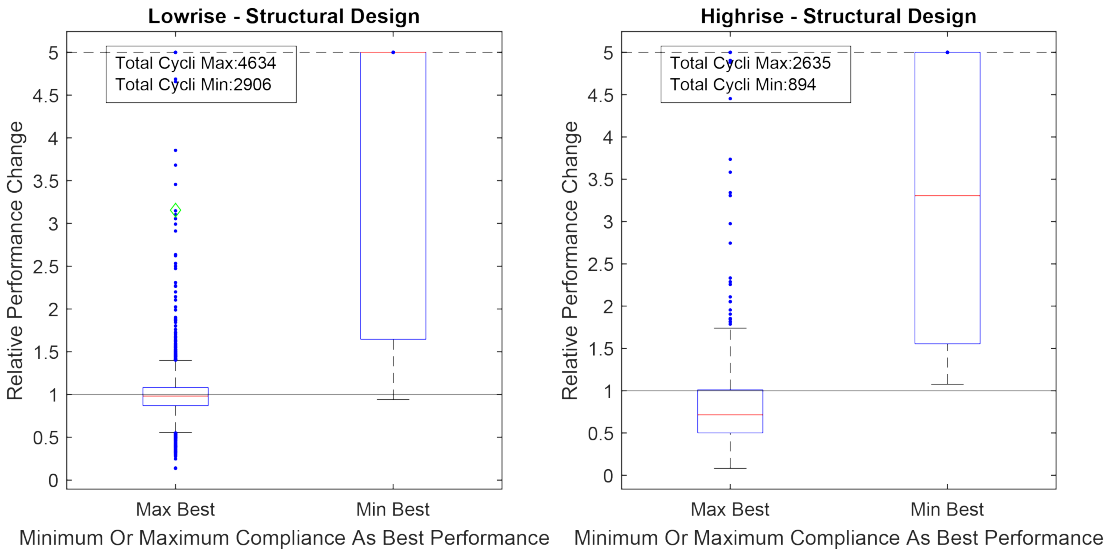


Figure 5.1: The three different designs used for the simulations

versus low compliance as the best performance result. It can be seen that assuming a high compliance as best performance results (on average) in building spatial designs that have an increased performance over the initial design. When examining the cycles of a low-compliance-best simulation it is evident why these result in decreased performances. The HBO approach removes the worst performing spaces, i.e. those at the bottom of the building spatial designs since they have the highest structural compliance. If and only if a complete level is removed, this approach resolves in an increased performance. If one or more spaces remain in the level, an overhang will be created increasing the structural compliance in the entire building.

Performance assessment for structural optimization should be performed by ranking spaces from highest to lowest compliance where the higher the compliance the better the performance. The basic principle behind this approach is to add material where high stress occur and to remove it where low stresses occur. This principle is in accordance with the research of Biyikli and To (2015), they directly use the strain to distribute the material for optimization, rather than using the density derivative to the strain and finding equal optimal solutions with less computational time.

Removal Options The results of the different removal techniques, as illustrated in Figure B.2, show that a more aggressive approach (30%, 40%, or 50%) leads to better performances than less aggressive approaches(1 space, 10%, or 20%). This is due to the optimal structural design will be found in low designs, i.e. a building spatial design one or two storeys high. The 50 % approach is more effective than the lower rates due to



(a) Lowrise Design *(b) Highrise Design*

Figure 5.2: Set 1 simulation results for three different building spatial designs optimized towards structural design. Values are normalized towards the performance of the initial design. For each design the high and low structural compliance results are shown.

the amount of spaces considered. When a 50% removal rate is applied over two cycles for a building of 90 spaces, relation between the amount of spaces removed and the total amount of spaces considered is $90 : 180 = 1 : 2$. For 40% removal three cycles are required giving a relation of $108 : 270 = 1 : 2.5$. Thus a higher amount of spaces are considered and the removal technique is less effective. Each step to a lower amount of removed and split spaces leads to an increase of spaces considered, with the largest results of removing one space each cycle with a relation of $90 : 900 = 1 : 10$. The results of the lowrise and cube design show less preference towards the 50% removal technique, due to the designs being three storeys high. Thus after the lower storey plus half of the second storey are remaining, the second cycle of removing 50% will thus remove spaces located on the lowest level. The removal options of 30% and 40% remove less per cycle thus only removing the top two storeys.

Modification Options Figure B.3 shows the simulation results per modification technique. It can be seen that for the cube and lowrise spatial design the scaling and sweeping modifications in the z direction are the most effective. This is due to the designs already being near the optimum elevation number and thus modifications that adjust space shapes become more effective. When a space is modified in z direction, the ratio between walls and floors change and since the loads on the walls are lower than the floors, this leads to a lower overall load on the building spatial design. If the total load on the building reduces, so does the compliance and thus the performance increases. Furthermore, it can be said that scaling in two directions is more effective for the cube than the one dimensional operations. Thus a square like shape is more effective than rectangular shapes. When considering a rectangular shape, like the lowrise design, modifications in the y directions are more effective than the x direction. This can also be seen as a preference of square shapes since the initial lowrise design is longer in x than y direction. It can be seen that the modifications operating only in z direction are less effective for the highrise design, this is not surprising since the building is supposed to reduce in height for optimal SD perfor-

mances. Another observation is that scaling operations are more effective than sweeping operations.

Splitting Options Figure B.4 presents the performances for the different splitting techniques, for SD optimization. The procedure that splits the largest space is the most effective splitting approach to increase the structural performance of a building spatial design. This is followed by splitting the best performing space, this is not unexpected since large spaces tend to have more compliance than small spaces and thus a better performance because of the larger spans. Splitting largest or best also supports the principle of adding material to locations where compliance is high, as mentioned in section 5.1.1 and Biyikli and To (2015), since it creates an additional wall in the middle of the space. Expanding this principle explains why splitting the smallest space is not as effective, since small spaces are stiffer and have less deformation if material is added there it is not in the most efficient place.

A subset of parameters can be made. In Removal options paragraph, it can be seen that the 50% removal technique is the most effective, for that parameter the results are filtered in Figure B.5. In these graphs, it can be seen that splitting the largest space is the most effective procedure to achieve a performance increase. That the parameters of splitting the best or worst space provide equal results can be explained straightforwardly as half the building design is removed. So the other half needs to be split and since only spaces can be split that have a performance indication all remaining spaces are split once. This is in contrast with splitting the largest space where a space can be split again if it is still the largest in a design.

5.1.2 Building Physics Optimization

Simulations using parameter set one do not consistently result in increased building physics performances, as illustrated in Figure B.6. The combination of individual assessment and removing the worst spaces in a building spatial design leads towards an approximation of a sphere. However, the BSO Toolbox operates in an orthogonal space and thus the optimal surface to volume ratio is a cubical design. Since the building physics performance is dependent on the ratio of surface area and volume of a spatial design, evolving a spatial design towards anything but a cubical shape will decrease its performance. Appendix D elaborates on a spherical approximation versus cubical shape in the orthogonal space. Acknowledging that the procedure of individual space assessment is in principle flawed for the building physics discipline, no further in depth analysis of different parameters has been carried out.

5.2 Simulations Using Parameter Set 2

Parameter set two focuses on the different assessment levels of spaces and includes the parameters as shown in table 5.2.

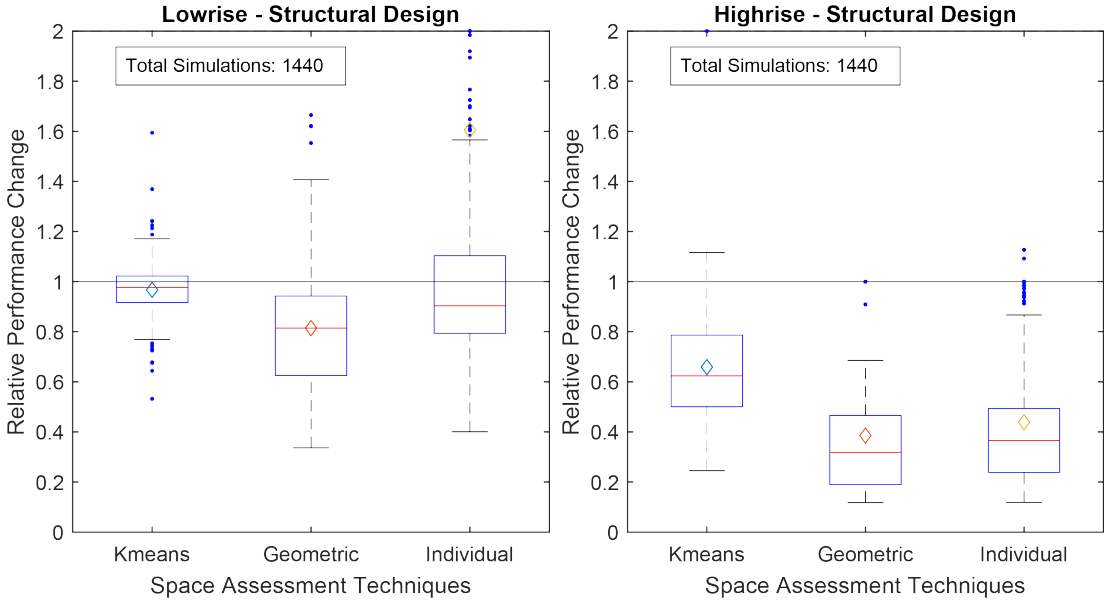
	Parameters
Performance Assessment	K-means, Geometrical Clustering, or Individual
Best Performance	SD: High, BP: Low
Removal Options	for individual assessment: 30 %, 40%, or 50 % for cluster assessment: 1 cluster
Number of cycles	So that the amount of spaces removed over all cycles equals at least the space count of the design
Scaling Options	X, Y, Z, XY, XZ, YZ, or XYZ
Sweeping Options	X, Y, Z axis at begin, middle or end
Splitting Options	Largest, Best, or Worst space
Splitting Preference	X-Y-Z, X-Z-Y, Y-X-Z, Y-Z-X, Z-X-Y, or Z-Y-X

Table 5.2: The parameters used in the second simulation set

5.2.1 Structural Optimization

The structural optimization is carried out with performances as normalized in equation 4.1a, i.e. the higher the structural compliance the better a space's performances is rated.

Performance Assessment Figure 5.3 show the results of different techniques to assess performances. Geometric clustering outperforms both the K-means clustering significantly and the individual techniques slightly in finding the more optimal performances. Figure B.8 shows the cycles needed for the procedure to optimize the highrise building spatial design. The average amount of cycles needed for geometric clustering is 6.8 cycles, whereas the average amount of cycles required for individual assessment is 2.0 cycles. With the size and complexity of the presented building designs a cycle takes around a minute to complete, thus being a time difference of 6.8 versus two minutes.



(a) Lowrise Design *(b) Highrise Design*

Figure 5.3: Performance assessment results for set 2 for structural performances. Only the best cycle of each parameter combination is considered.

Modification Options Since parameter set 1 focused on the individual assessment of spaces, and the K-means algorithm did not prove to be effective, for this section a subset is filtered and includes only the solutions of geometrical clustering. The results for the modification techniques of geometrical clustering are illustrated in Figure B.9. Each of the modification techniques is capable of consistently improving a building spatial design's performance. Some difference can be observed in the scaling graphs between the techniques that scale over a single axis and those that scale over multiple axis. Even for the lowrise design which is stretched in x direction, thus scaling in y direction intuitively seems as a more effective method. Nevertheless, scaling over simultaneously x and y axis reaches a better solution than scaling over either x or y axis. The sweeping techniques over the y dimension does outperform those which modify over the x dimension, with this modification only a couple of spaces are altered and most of the spaces are still in their cubical shape. While all solutions are within the stated constraints of space count and volume, not all results are desired spatial designs. Especially with modifications over the z direction spaces take shapes which are unrealistic for a normal space size, e.g. very high spaces compared to their width and/or depth as Figure 5.4 illustrates. Section 5.3 will elaborate more on the topic of unrealistic building designs.

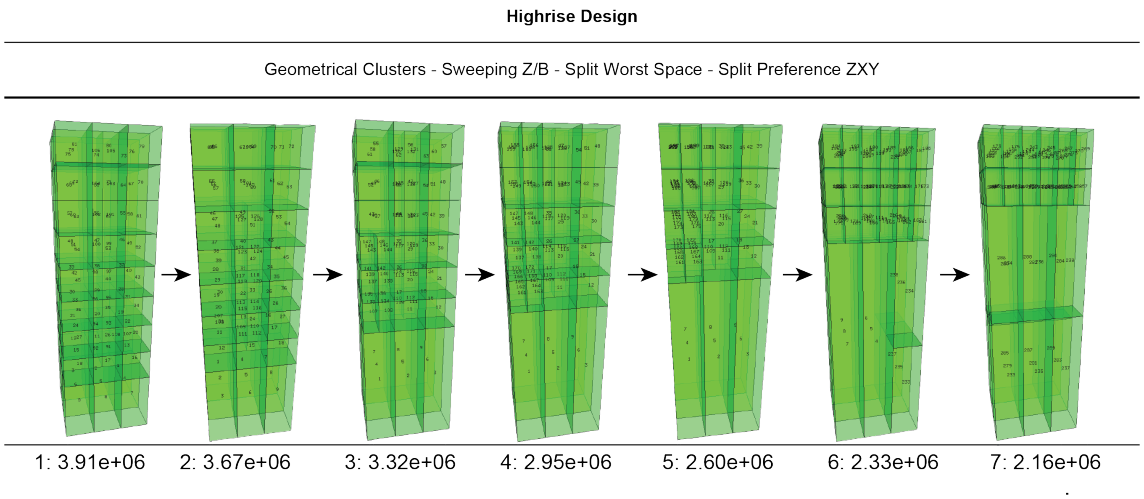


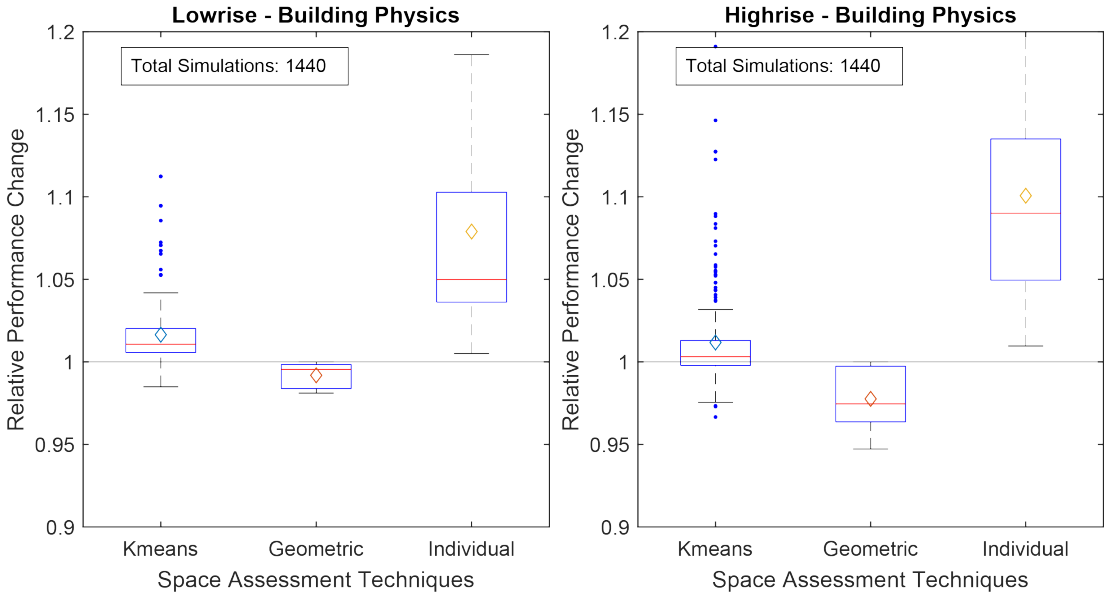
Figure 5.4: A simulation of co-evolutionary building spatial design optimization for the highrise design for the structural discipline. As the optimization goes further in cycles the building spatial design becomes more unrealistic with spaces of 3m wide, 3m deep and 18m high.

Splitting Options The results for the splitting options are illustrated in Figure B.10. It can be observed that splitting the worst space results for both the cube and lowrise design results in the best solutions. However, just as in section 5.2.1 these splitting result in spaces that have an undesirable height to width/depth relation, as Figure 5.4 illustrates. The other two options, split largest or best, have averages that are about equal and are both capable of improving a building spatial design's structural performance. Figure B.11 shows the difference in splitting preference.

5.2.2 Building Physics Optimization

In the following paragraphs the results of optimization of the three designs towards the building physics discipline using parameter set 2 will be shown and elaborated.

Performance Assessment Figure 5.5 shows the results of the performance assessment techniques. The individual assessment approach does not lead to improved building physics performances, in accordance with parameter set 1, see section 5.1.2. The K-means techniques is comparable to the individual approach in such a manner that it groups equal performing spaces. Thus instead of removing the corner spaces one-by-one, they are all removed as one cluster and again moving to a spherical approach. The geometrical cluster technique does result in increased building physics performances for all three designs. With the approach of considering a facade as one cluster, designs are capable of retaining the square shapes and smooth facades in the overall spatial design.



(a) Lowrise Design *(b) Highrise Design*

Figure 5.5: Performance assessment results for simulation set 2 for building physics performances. Only the best cycle of each parameter combination is considered.

Modification Options Figure B.13 shows the results of the modification techniques focused on geometrical clustering. The most effective modifications are those that steer a design towards a cubical shape. Improvements for the cube design are minimal, the highest improvement is 0.4%. While for the highrise design improvements of over 5 % are found. The lowrise design profits most from modifications over the y axis, which is logical since the building is directed in x direction thus the building spatial design evolving into a square shape. For the highrise design the scaling over x and y axis combined is the most effective approach. Except for modifications over only the z axis, all modification techniques show the ability of improving a building physics design towards the BP discipline.

Splitting Options Figure B.14 show the results for the different split techniques for geometrical clusters. For the cube and lowrise design the difference between the parameters is small. Figure B.15 shows that their are small differences between the preferred splitting dimensions. That differences are small is most likely due to that the best solutions are found after multiple iterations, so the procedure has had the opportunity to split all dimensions.

5.3 Unrealistic Building Spatial Designs

While certain parameters are capable of improving the performances of the initial building spatial designs, not all solutions are desirable from an AEC engineer's point of view, see Figure 5.4. To investigate this, all simulations of parameter set 2 are checked for the requirements given below.

- If a space's height is more than the average of the width and depth, and the space's height is at least 3000 mm.
- If a space's width is more than 20 times the depth.
- If a space's depth is more than 20 times the width.

These rules are set in order to maintain some kind of realistic shapes to spaces. The height is related to the width and depth of a space, and not limited to a certain maximum height. While not all spaces are required to be high, certain spaces require to be higher than the 3 meters of a standard work or living environment, i.e. foyers, theater stages, shops, etc. When a building spatial design is checked with the requirements it's level of unrealistic spaces is also calculated. Let μ be the ratio of spaces that are unrealistic versus the total amount of spaces in a design, and ν the set of designs that contain one or more unrealistic spaces. The level of unrealistic spaces (L.U.) is defined as the average μ for each design in ν.

$$\mu = \frac{\text{amount of unrealistic spaces}}{\text{total amount of spaces}} \tag{5.1}$$

Table 5.3 presents the values of unrealistic designs for the simulations in parameter set 2. All even columns show the percentage of unrealistic building for those parameters, all uneven columns shows the level of unrealistic spaces, with 0 being completely unrealistic and 1 being realistic. When calculating the average percentages for performance evaluation it shows that 38% of all structural and 42% of all building physics optimizations result in unrealistic buildings. From the table it is also clear that modifications which operate in the z direction have a large impact on these numbers. Modifications over x and y almost never result in unrealistic designs. The values for splitting dimension preference are consistent for each design and discipline, showing that these parameters all have an equal contribution to unrealistic building spatial designs.

	Cube				Lowrise				Highrise			
	SD %	SD L.U.	BP %	BP L.U.	SD %	SD L.U.	BP %	BP L.U.	SD %	SD L.U.	BP %	BP L.U.
Performance Assessment												
Indi	38.0	0.083	43.0	0.072	39.0	0.076	41.0	0.15	37.0	0.24	42.0	0.029
K-means	44.0	0.17	44.0	0.23	43.0	0.22	44.0	0.29	41.0	0.33	44.0	0.24
Geo	30.0	0.14	37.0	0.052	31.0	0.091	44.0	0.42	35.0	0.38	38.0	0.34
Scaling Modification												
X	0	0	0	0	0	0	0	0	0	0	0	0
Y	0	0	0	0	0	0	0	0	0	0	0	0
Z	100.0	0.14	100.0	0.11	80.0	0.16	80.0	0.18	80.0	0.31	80.0	0.09
XY	0	0	0	0	0	0	0	0	0	0	0	0
XZ	100.0	0.12	100.0	0.13	97.0	0.044	100.0	0.34	93.0	0.53	100.0	0.24
YZ	100.0	0.11	100.0	0.11	100.0	0.11	90.0	0.53	93.0	0.56	100.0	0.25
XYZ	92.0	0.079	95.0	0.2	90.0	0.15	93.0	0.29	87.0	0.36	100.0	0.25
Sweeping Modification												
X/B	0	0	0	0	0	0	0	0	0	0	0	0
X/M	0	0	0	0	0	0	0	0	0	0	0	0
X/E	0	0	0	0	0	0	8.9	0.003	0	0	0	0
Y/B	0	0	0	0	0	0	0	0	0	0	0	0
Y/M	0	0	0	0	0	0	0	0	0	0	0	0
Y/E	0	0	0	0	0	0	0	0	0	0	0	0
Z/B	69.0	0.1	92.0	0.057	87.0	0.13	100.0	0.11	79.0	0.069	93.0	0.044
Z/M	69.0	0.15	92.0	0.053	80.0	0.11	100.0	0.077	81.0	0.074	93.0	0
Z/E	69.0	0.11	92.0	0.071	79.0	0.09	100.0	0.14	83.0	0.032	93.0	0.027
Splitting Options												
Large	36.0	0.089	40.0	0.086	35.0	0.12	41.0	0.22	31.0	0.34	39.0	0.15
Best	41.0	0.079	43.0	0.12	41.0	0.08	43.0	0.23	39.0	0.3	42.0	0.13
Worst	36.0	0.19	43.0	0.11	39.0	0.14	42.0	0.25	42.0	0.24	42.0	0.11
Splitting Dimension Preference												
XYZ	38.0	0.11	42.0	0.15	39.0	0.12	42.0	0.31	38.0	0.35	41.0	0.17
XZY	38.0	0.1	42.0	0.14	39.0	0.07	43.0	0.19	39.0	0.25	41.0	0.13
YXZ	38.0	0.12	42.0	0.11	39.0	0.12	42.0	0.32	39.0	0.38	41.0	0.17
YZX	38.0	0.11	42.0	0.092	38.0	0.091	42.0	0.26	38.0	0.29	41.0	0.13
ZXY	37.0	0.12	41.0	0.067	38.0	0.14	42.0	0.18	35.0	0.23	41.0	0.11
ZYX	37.0	0.13	41.0	0.073	38.0	0.14	42.0	0.14	35.0	0.22	41.0	0.071

Table 5.3: Every even columns shows the percentage of unrealistic building designs for the parameters given. Every uneven column shows the level of unrealistic(L.U.) spaces in those unrealistic designs, with 1 being a complete unrealistic design and 0 being a realistic design.

6 Discussion

This study focuses on the extension of a heuristic optimization tool for early stage building spatial designs. It shows that heuristics optimization procedures are capable of improving a building spatial design for structural or building physics disciplines, evolving it during multiple cycles and increasing its performance. While a multitude of parameters have been implemented, many more can be developed. The study as shown provides an early exploration of basic modifications and rules to learn from and get inspired by to find and develop new heuristic procedures. The Heuristic Building Optimization toolbox offers an infrastructure to accommodate future research and development. This chapter discusses multiple topics which are up for debate and should be taken into consideration while reading this thesis.

Multi-Disciplinary Optimization Due to the vastness and complexity of the parameter sets developed in this research no elaboration of multi-disciplinary optimization has been carried out. An understanding of individual disciplines is necessary to ensure procedures are, first of all, capable of optimizing at all. That not all the presented procedures are capable of optimizing certain disciplines can be seen with the individual assessment of spaces not being able to optimize for the building physics discipline.

Constraints The question if all solutions are desirable can be asked. Multiple parameter combination show that structural performances of buildings can increase due to a decreasing floor area. That this is favorable for the resulting performance follows from the fact that less floor area, gives less load on the structure and thus less structural compliance. Two major parameters involved in these transformations are the modifications in height and the splitting of the worst spaces in a design. While the study aims to improve performances given a constant volume and space count, in the architecture, engineering and construction industry the amount of floor area is largely of importance in determining the monetary value of a building spatial design. Thus the argument can be made that the volume constraint is not practical for building spatial design optimization, and a better constraint would be floor area of a building spatial design. Another possibility can be found in the addition of a new discipline for the BSO Toolbox focused on (monetary) value of spaces and buildings, so if the floor area of a building decreases its real estate performance decreases and the solution is worse than its predecessor.

Simulation Settings The structural and building physics properties are held constant during this study to enable a comparison between the parameters introduced in the heuristic procedure. Altering the load cases, material properties, and constraints might lead to different solutions and insights. Especially since the ratio between floor and wall loading proved to be relevant in this study.

Performance Assessment The K-means algorithm used in this study is not optimized for these problems. It clusters spaces using a predefined maximum amount of clusters and iterations after which it finds the optimal amount of clusters and distribution of spaces over those given the provided settings. However, these settings might not result in the global optimum for the clustering of building spatial designs. For geometrical clustering only the outer layer of a facade is added to a cluster, however, adding multiple layers of spaces to clusters might make the approach more powerful and efficient since less iterations will be necessary to reach the optimal solution.

Space Removal For structural optimization, it can be stated that highly aggressive removal approaches are more effective in moving towards a more optimal solution than low aggressive removal approaches. This goes for both the quality of the found solution as well as the amount of cycles required to reach it. Optimizing a building spatial design towards the building physics discipline was only achieved after considering geometrical clusters, those are only capable of removing one cluster at a time for the current implementation.

Modifications In the sweeping procedure three locations(begin, middle, end) are currently implemented. It should be obvious that more options are possible at any location in the building, or using specific geometry properties of the building spatial design or spaces in it.

Splitting Spaces are always split over their largest dimension in this study. The optimal shape, for designs with a single space, for building physics is cubical and for structural design it is a building with a square floor plan. In order to achieve these shapes it is required to split the largest dimension. However, when constructing building spatial designs out of multiple spaces splitting over other dimensions might increase performances. For structural design this is mainly due to the increase in structural volume that comes with non-square floor plans, a square has the optimal ratio between circumference and area.

7 Conclusions

The goal of this graduation project is to *develop heuristic rules for the optimization of early stage building spatial designs.* In order to answer this goal, two sub questions have been formulated namely: 1: *How to interpret performance evaluations of a early stage building design?*, and 2: *How to use that interpretation to modify the early stage building spatial design such that its performance improves?* For the first question it can be concluded that the performances of a building spatial design can be evaluated with either taking a high or low energy as best performance for a space, which should be chosen is dependent on the nature of the discipline optimized. The research shows that the structural discipline should be evaluated with a high (strain) energy as best, i.e. spaces are rated from high to low (strain) energy as best to worst performance. Thus the procedure removes material at locations where it is not fully utilized and add material at locations where a high compliance is. This principle is in accordance with the proportional topology research of Biyikli and To (2015). The building physics discipline should be evaluated with a low (thermal) energy (loss) as best, i.e. spaces are rated from low to high (thermal) energy (loss) as best to worst performance. Thus the procedure removes the spaces with most facade area, while keeping the spaces that are surrounded by others.

When the performances are set they are assessed with different procedures. For structural design all three assessment techniques (K-means clusters, geometrical sets, Individual Spaces) are capable of improving the initial spatial design. Of these three techniques, geometrical sets result in solutions with the best performances. However, the computational time that is required for geometrical sets is higher than for the other techniques, up to six times of the approach with individual spaces. Optimizing for the building physics discipline is only possible with the geometrical cluster technique, both the K-means and the individual approach do not consistently lead to an improvement for building physics performances. The difference between the techniques can be found in the type of geometrical object the technique wants to approach. For the individual and K-means technique

the optimization moves towards a spherical shape, whereas the geometrical technique let the optimization move towards a cubical shape. A perfect sphere has the ideal surface to volume ratio. However, the building spatial optimization toolbox is developed in an orthogonal space, thus spherical shapes can only be approximated and the ideal surface to volume ratio is found in the cubical shape.

The second question has several parameters involved. The first parameter concerns the amount of spaces removed in each cycle and the amount of cycles required to reach the best solution of that parameter. For structural design it can be stated that, when using the individual assessment, a more aggressive removal approach (30%, 40%, or 50% of spaces removed) leads to better performances than the less aggressive approaches (one space, 10%, or 20% of spaces removed). Besides leading to a better result, the aggressive approaches are also more effective, i.e. they reach their optimal solution in less cycles. Since the individual assessment of spaces does not lead to improved solution for building physics optimization, the individual removal percentages also do not result in improved solutions. Removal in combination with cluster assessment is done by removing one cluster at the time. For structural optimization removing a cluster of either K-means or geometrical procedure result in more optimal solutions. For building physics optimization removing a cluster constructed with the K-means procedure does not lead into improved solutions, removing a cluster constructed by the geometrical procedure does result into improved solutions.

After spaces are removed from the building spatial design the remaining spaces are modified to meet the volume constraint set. The most optimal modification techniques for structural heuristic optimization are those that reduce the total height of a building spatial design and keep the floor plan of the building square. Several improved solutions are found which included modification of the height of a design. However, they alter the total amount of load applied to the building by reducing the total floor area while keeping the total surface area equal to the initial design. So while the results are within the boundaries set for this research, the question arises whether these results are wanted. The most optimal techniques for building physics optimization are those that move the building spatial design towards a cubical shape. For the lowrise design that can be observed with the most effective method being modifications in the y direction. The highrise design profits most from modifications over x and y simultaneously. Both lowrise and highrise benefit from other modification techniques as well, steering the designs more towards a cuboid shape.

The four different techniques (split largest, smallest, best, or worst) to split a space are capable of improving a building spatial design's performance. However, the procedures of splitting the smallest or worst space result in a design where a set of small spaces and a couple of large spaces are evolved. This leads to a reduction of total floor area when combined with modifications in height direction. For the preference of which dimension to split little difference can be found between the six options (preference X-Y-Z, X-Z-Y, Y-X-Z, Y-Z-X, Z-X-Y, or Z-Y-X). This is mainly due to the fact that the procedure does not find the optimal solution within a single cycle, and if the procedure runs over multiple cycles the other dimensions will be split.

8 Recommendations

The presented heuristic building optimization procedure shows that it is capable of optimizing building spatial design given the constraints and parameters provided. However, a part of the solutions in the design search space are not desirable due to the geometric

properties and relations of spaces. Following research should reconsider the constraints and definitions as stated in the presented research. A shift towards floor area constraints instead of volume constraint might be a more preferable tool from an engineer's point of view.

The calculation of structural performances is another topic up for debate, currently the structural performance of a space is the summation of the strain energy of all structural elements of a space. When the stiffness of one wall would be lowered, e.g. due to windows, the strain energy in the space would rise but whether the space should have a better performance is uncertain. A possible solution to deal with this situation might be to set a strain energy relative to the amount of structural volume in a space.

Another structural adjustment is the determination of maximum deflection. Currently, the amount of deflection is not reviewed and taken into account while the size of most real structural elements are determined by the maximum allowable deflection. Especially when multiple materials are used in the structural models. A space with stiffer or thicker material might have a higher amount of strain energy over a space with less stiff or thinner material, while the later space might have a larger deformation and thus would benefit from adding additional material, i.e. splitting the space.

The procedure for geometrical clustering can also be improved and evolved. Instead of searching for the outer layer of spaces a range can be set, i.e. 20 % of the width, all spaces within this range are clustered. This might allow for a more effective approach in the sense of computational power.

A completely new approach might be to keep the global shape of a building spatial design intact, and altering the configuration of spaces within the building spatial design, e.g. for structural design it might be beneficial to merge spaces at the location of low compliance and splitting spaces at the location of high compliance.

The structural loads in this study's settings are completely symmetrical. However, governing load combinations in real structures might include asymmetrical load such as snow, wind, or life loading. These data are location specific but might result in different optimal results. The building physics optimization of the cube design showed that the optimal shape is near cubical but due to the ground temperature the height is slightly smaller than the width and depth. Altering the ground profile might result in a different optimal design.

The heuristic procedure as developed allows for multidisciplinary optimizations. However, no multidisciplinary study has yet been conducted since the procedure had to be applied to single discipline optimizations first. A follow up study can make use of the infrastructure created and the parameters already investigated to execute multidisciplinary optimizations.

Acknowledgments

The author wishes to express his gratitude towards H. Hofmeyer, S. Boonstra, and B. d. Vries for their dedication, enthusiasm, and support during the supervision and development of this research. Th. de Goede for his input, criticism and perspective on all aspect concerning this project. Thanks to friends and fellow students for their questions and interest, which made me critically think about formulations and explanations. And special thanks to my family who fully supported and enabled this graduation project to be made and finished.

References

Adamski, M. (2007). Optimization of the form of a building on an oval base. *Building and Environment*, *42*(4), 1632 - 1643.

Akin, . (2001). Chapter 6 - variants in design cognition. In C. M. Eastman, W. M. McCracken, & W. C. Newstetter (Eds.), *Design knowing and learning: Cognition in design education* (p. 105 - 124). Oxford: Elsevier Science.

Ball, G. H., & Hall, D. J. (1965). Isodata, a novel method of data analysis and pattern classification.

Bichiou, Y., & Krarti, M. (2011, 12). Optimization of envelope and hvac systems selection for residential buildings. *Energy and Buildings*, *43*, 3373-3382.

Biyikli, E., & To, A. C. (2015, 12). Proportional topology optimization: A new non-sensitivity method for solving stress constrained and minimum compliance problems and its implementation in matlab. *PLOS ONE*, *10*(12), 1-23.

Boonstra, S. (2016). Multi-disciplinary optimisation (master thesis)'. *TU/e Repository*.

Boonstra, S., van der Blom, K., Hofmeyer, H., Emmerich, M. T. M., van Schijndel, A. W. M., & de Wilde, P. (2018, 3 20). Toolbox for super-structured and super-structure free multi-disciplinary building spatial design optimisation. *Advanced Engineering Informatics*, *36*, 86–100.

Bouchlaghem, N., & Letherman, K. (1990). Numerical optimization applied to the thermal design of buildings. *Building and Environment*, *25*(2), 117 - 124.

Cheung, C., Fuller, R., & Luther, M. (2005). Energy-efficient envelope design for high-rise apartments. *Energy and Buildings*, *37*(1), 37 - 48.

Chow, T., Zhang, G., Lin, Z., & Song, C. (2002). Global optimization of absorption chiller system by genetic algorithm and neural network. *Energy and Buildings*, *34*(1), 103 - 109.

D'Cruz, N. A., & Radford, A. D. (1987). A multicriteria model for building performance and design. *Building and Environment*, *22*(3), 167 - 179.

der Merwe, D. W. V., & Engelbrecht, A. P. (2003, Dec). Data clustering using particle swarm optimization. In *Evolutionary computation, 2003. cec '03. the 2003 congress on* (Vol. 1, p. 215-220 Vol.1).

European Construction Tehnology Platform. (2005). Challenging and changing europe's built environment: A vision for a sustainable and competitive construction sector by 2030.

Flager, F., & Haymaker, J. (2007). *A comparison of multidisciplinary design, analysis and optimization processes in the building construction and aerospace industries.*

Flager, F., Welle, B., Bansal, P., Soremekun, G., & Haymaker, J. (2009). Multidisciplinary process integration and design optimization of a classroom building. *Journal of Information Technology in Construction (ITcon)*, *14*(38), 595–612.

Friess, W. A., Rakhshan, K., Hendawi, T. A., & Tajerzadeh, S. (2012). Wall insulation measures for residential villas in dubai: A case study in energy efficiency. *Energy and Buildings*, *44*, 26 - 32.

Fuyama, H., Law, K., & Krawinkler, H. (1997). An interactive computer assisted system for conceptual structural design of steel buildings. *Computers And Structures*, *63*(4), 647 - 662. (Computing in Civil and Structural Engineering)

Haymaker, J., Kunz, J., Suter, B., & Fischer, M. (2004). Perspectors: Composable, reusable reasoning modules to automatically construct a geometric engineering view from other geometric engineering views. *Advanced Engineering Informatics*, *18*, 49-67.

Hofmeyer, H., & Davila Delgado, J. M. (2013). Automated design studies: Topology versus one-step evolutionary structural optimisation. *Advanced Engineering Informatics*, *27*(4), 427 - 443.

Hofmeyer, H., & Davila Delgado, J. M. (2015). Coevolutionary and genetic algorithm based building spatial and structural design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, *29*.

Hofmeyer, H., & Emmerich, M. T. M. (2013). Excellent buildings using forefront mdo: Lowest energy consumption, optimal spatial and structural performance. *NWO-NTT Proposal*. (For more information: https://www.nwo.nl/onderzoek-en-resultaten/programmas/open+technologieprogramma/projecten/2015/2015-13596, accessed on 18-09-2019)

Horváth, I. (2005). *On some crucial issues of computer support of conceptual design* (D. Talabă & T. Roche, Eds.). Dordrecht: Springer Netherlands.

Jain, A. K. (2010). Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, *31*(8), 651 - 666. (Award winning papers from the 19th International Conference on Pattern Recognition (ICPR))

Jedrzejuk, H., & Marks, W. (2002). Optimization of shape and functional structure of buildings as well as heat source utilisation. partial problems solution. *Building and Environment*, *37*(11), 1037 - 1043.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE transactions on information theory*, *28*(2), 129–137.

Machairas, V., Tsangrassoulis, A., & Axarli, K. (2014). Algorithms for optimization of building design: A review. *Renewable and Sustainable Energy Reviews*, *31*, 101–112.

MacQueen, J., et al. (1967). Some methods for classification and analysis of multivariate observations. *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, *1*(14), 281–297.

Marks, W. (1997). Multicriteria optimisation of shape of energy-saving buildings. *Building and Environment*, *32*(4), 331 - 339.

Okudan, G. E., & Tauhid, S. (2008). Concept selection methods–a literature review from 1980 to 2008. *International Journal of Design Engineering*, *1*(3), 243–277.

Rezaee, R., Brown, J., Haymaker, J., & Augenbroe, G. (2018, 08). A new approach to performance-based building design exploration using linear inverse modeling. , 1-27.

Ritter, F., Geyer, P., & Borrmann, A. (2013). The design space exploration assistance method: constraints and objectives.

Shea, K., Aish, R., & Gourtovaia, M. (2005). Towards integrated performance-driven generative design tools. *Automation in Construction*, *14*(2), 253 - 264. (Education and Research in Computer Aided Architectural Design in Europe (eCAADe 2003), Digital Design)

Steinhaus, H. (1956). Sur la division des corp materiels en parties. *Bulletin de l'academie polonaise des sciences*, *1*(804), 801.

Torres, S. L., & Sakamoto, Y. (2007). *Facade design optimization for daylight with a simple genetic algorithm.*

Verbeeck, B., Van Steirteghem, J., De Wilde, W. P., & Samyn, P. (2005). The need for numerical techniques for the optimization of structures using morphological indicators. *Computer Aided Optimum Design in Engineering IX*, 65-72.

A Overview HBO Toolbox

This appendix gives an graphic overview of the procedures in the HBO Toolbox as developed for this research. Illustrated in Figure A.1 is the loop which the HBO optimization follows. The procedure of Performance Calculation is not elaborated since it was developed in earlier research. For Performance Evaluation, Space Ranking, and Building Modification the different choices and parameters are elaborated in Figures A.2, A.3, and A.4 respectively

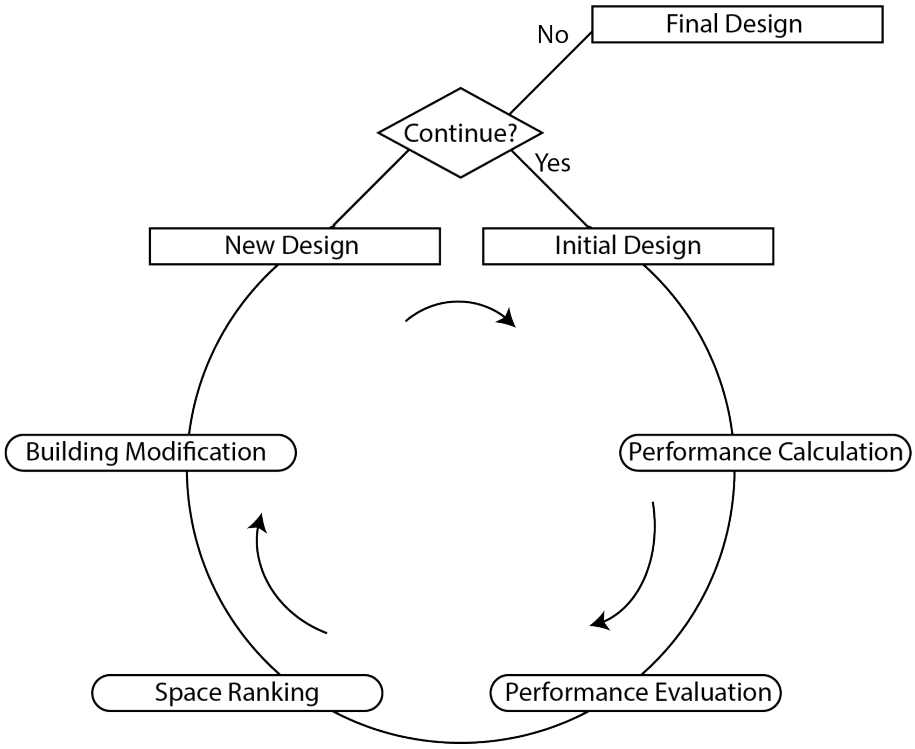


Figure A.1: The optimization loop of the heuristic building optimization procedure.

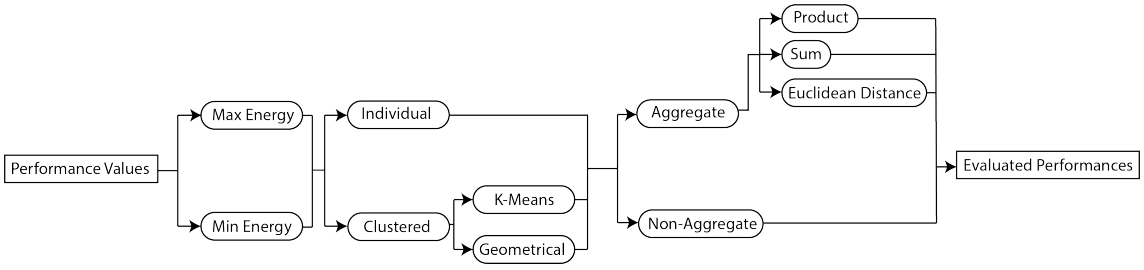


Figure A.2: All choices and possibilities developed to evaluate performances

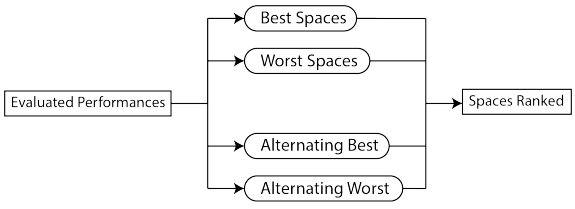


Figure A.3: All choices and possibilities developed to rank spaces

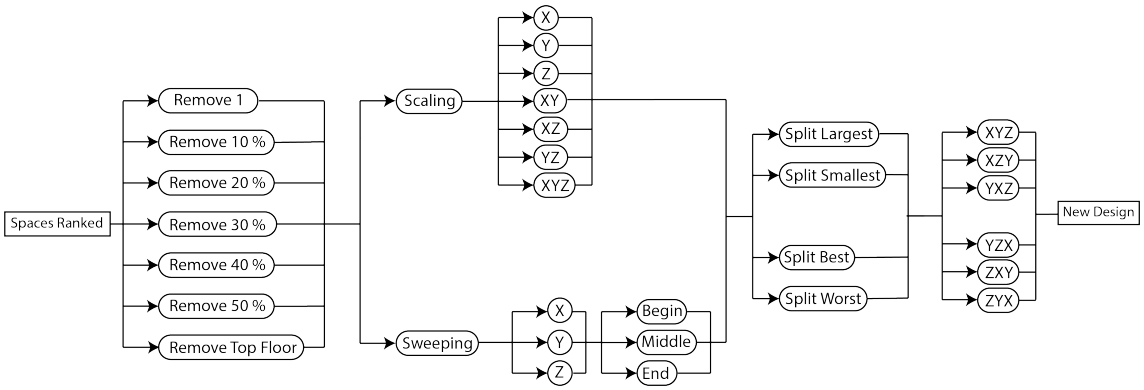


Figure A.4: All choices and possibilities developed to modify building spatial designs

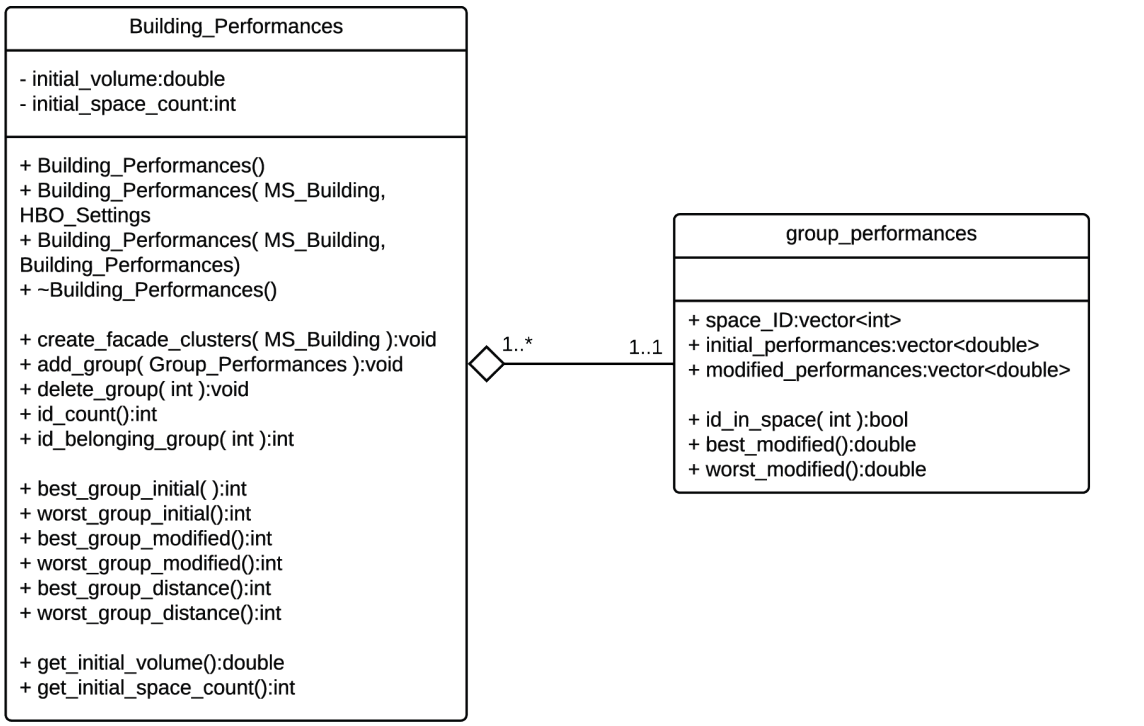


Figure A.5: UML class diagram of the Building Performances and Group Performances classes as developed for this research

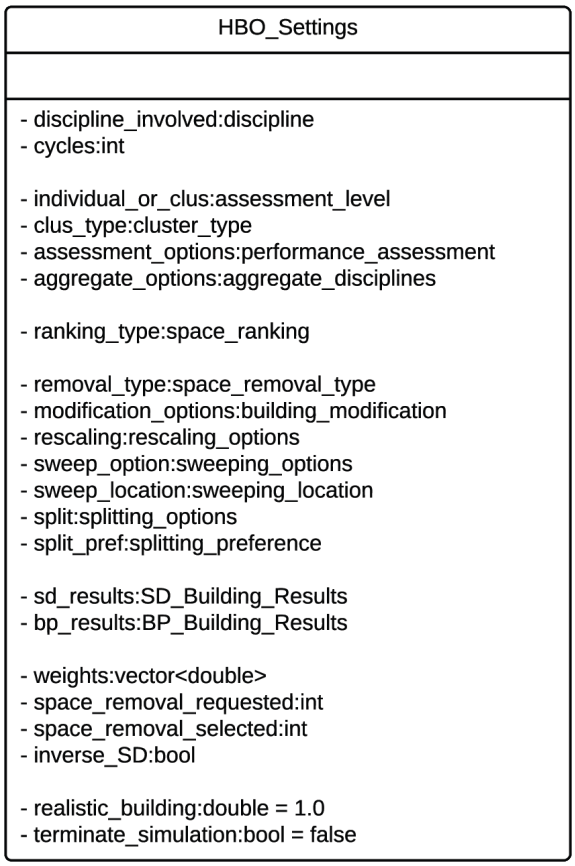
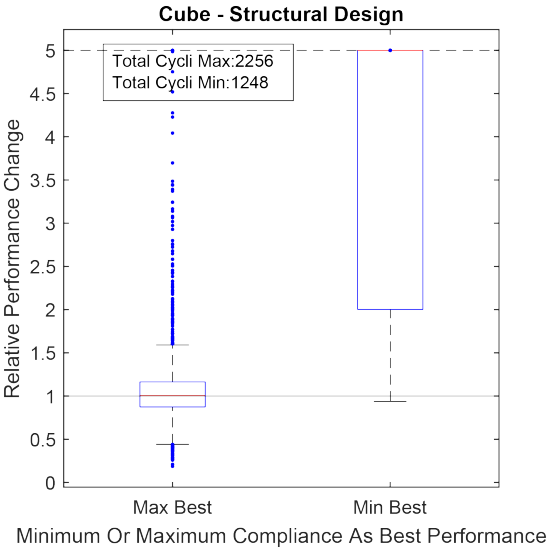


Figure A.6: UML class diagram of the settings structure as developed for this research

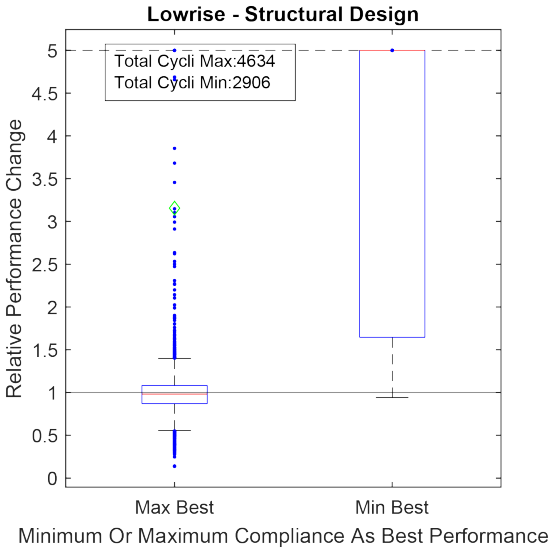
B Simulation Results

This appendix contains the boxplots with the simulation results.

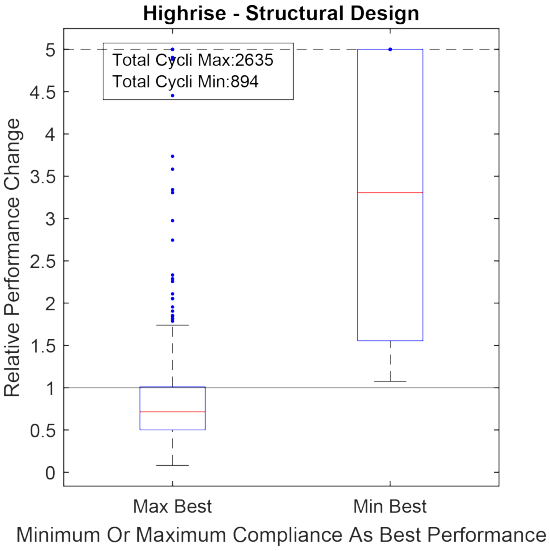
B.1 Parameter set 1



(a) Cube Design

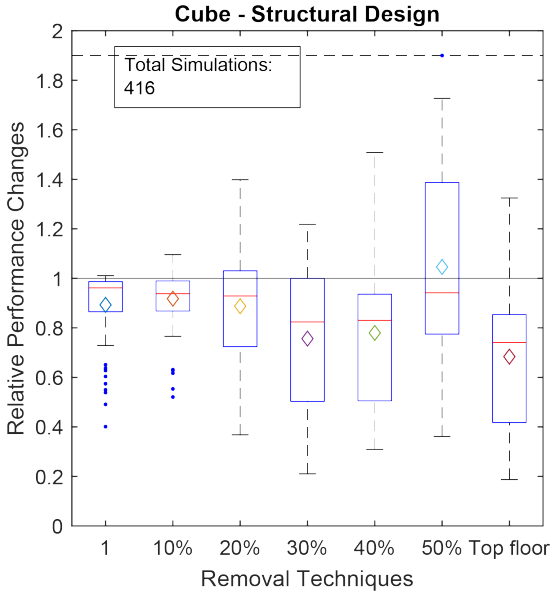


(b) Lowrise Design

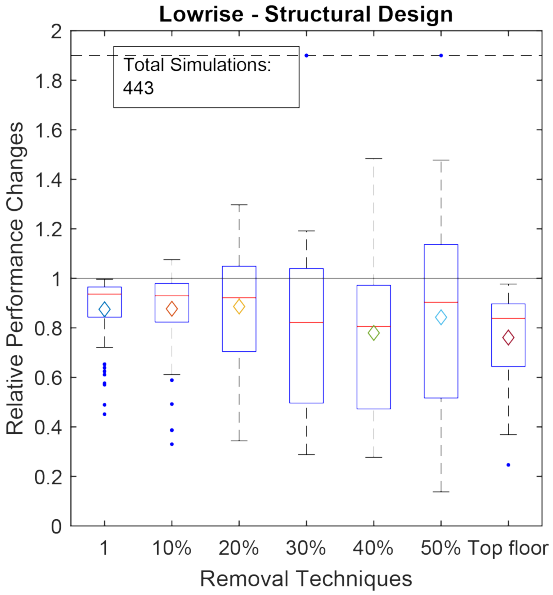


(c) Highrise Design

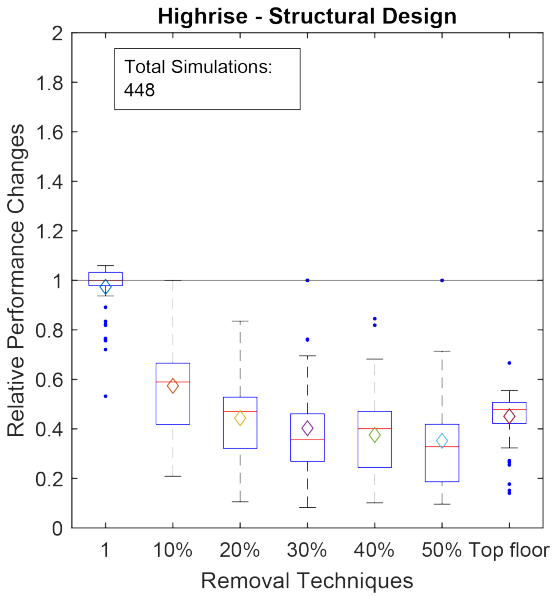
Figure B.1: Simulation results from set 1 for three different building spatial designs optimized towards structural design. Values are normalized towards the performance of the initial design. For each design the high and low structural compliance results are shown.



(a) Cube Design



(b) Lowrise Design



(c) Highrise Design

Figure B.2: Normalized simulation results for three different building spatial designs. Normalized values are calculated towards the performance of the initial design. For each design the difference between the amount of removed spaces is shown.

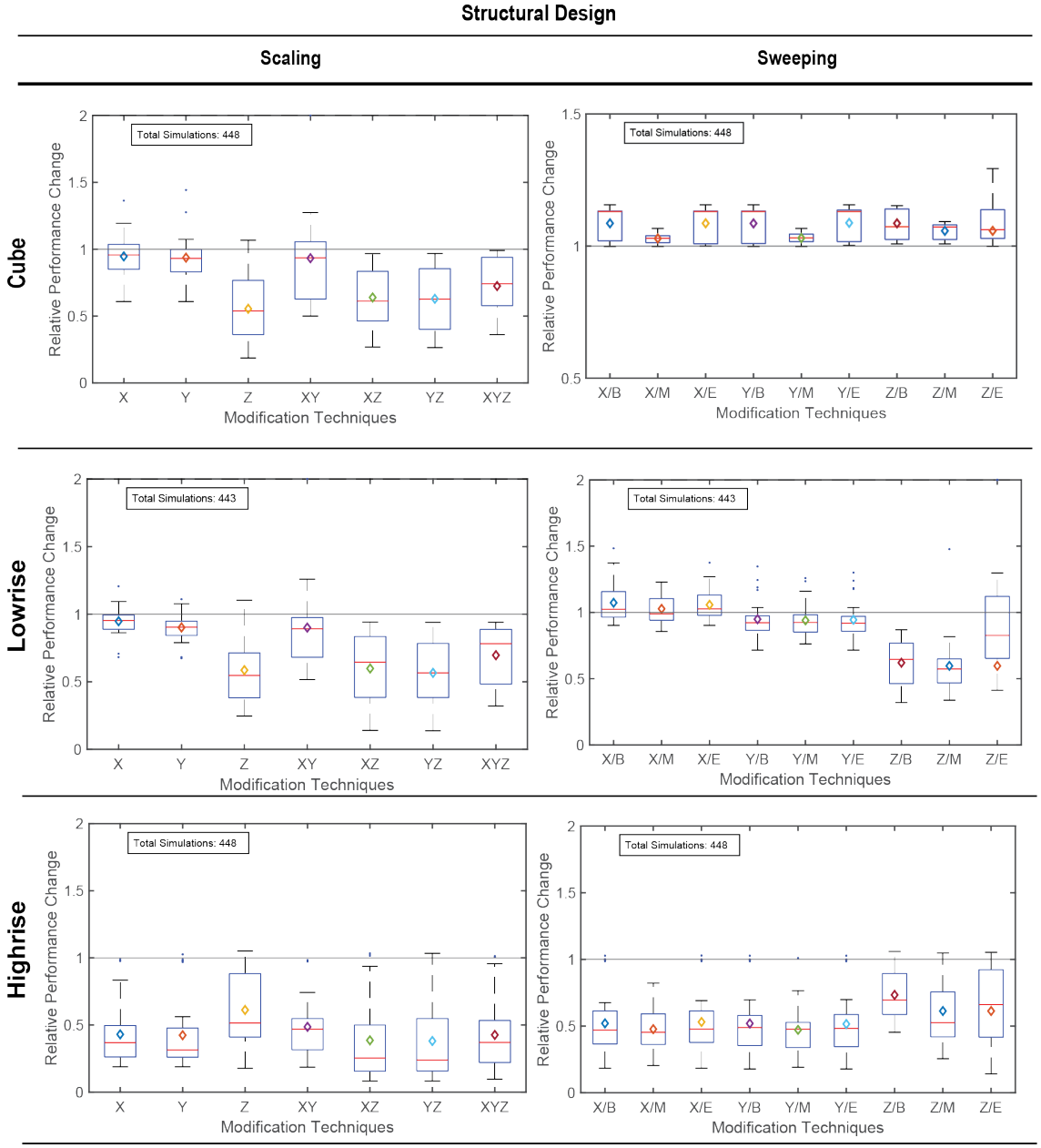
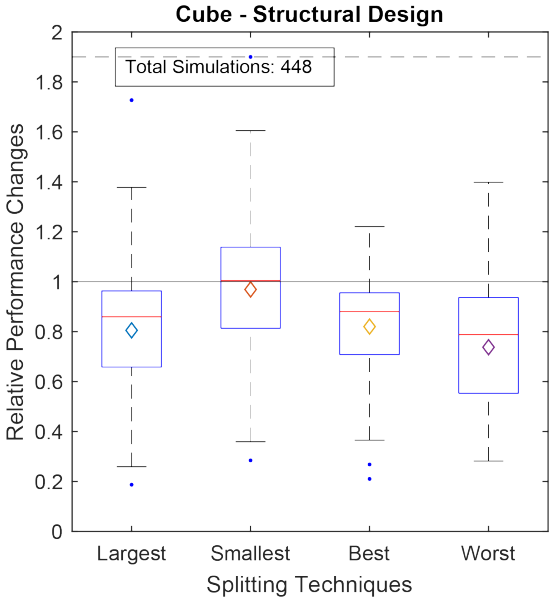
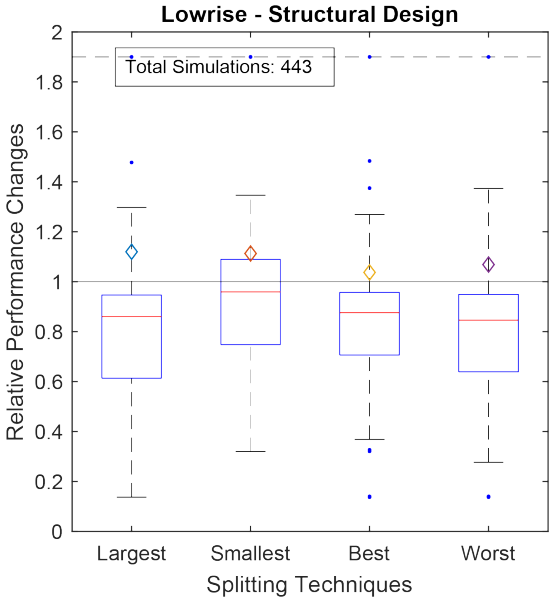


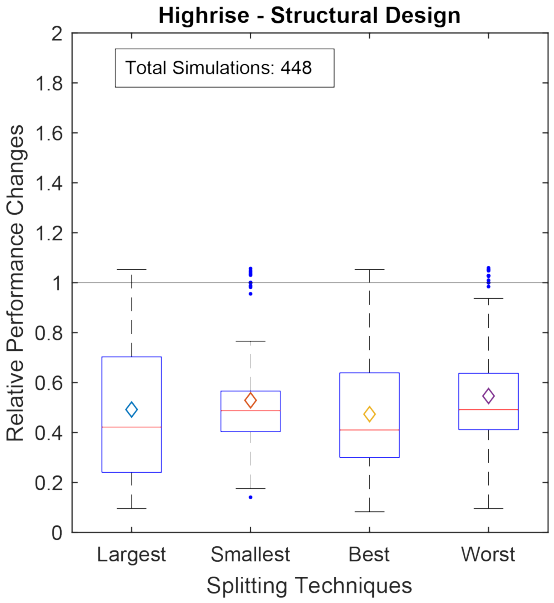
Figure B.3: Scaling and sweeping modification results for structural performances for set 1. For scaling the horizontal axis describes over which axis the building spatial design is scaled. The horizontal axis for sweeping consist out of two elements: the first describes over which axis the sweep is executed, the second at which location is the building spatial design with B, M, and E for begin, middle, and end respectively.



(a) Cube Design

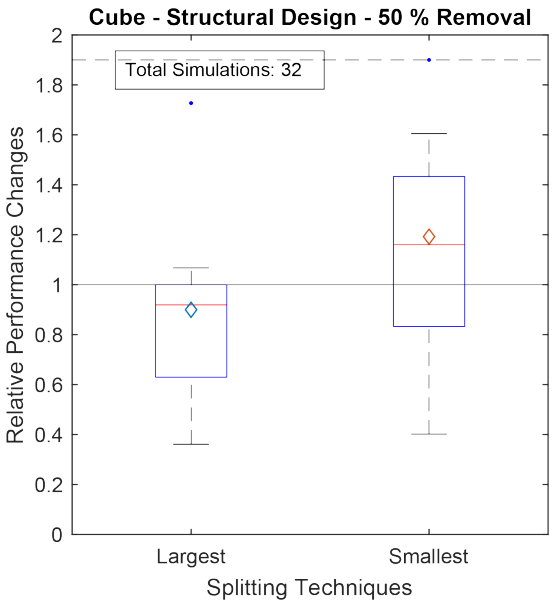


(b) Lowrise Design

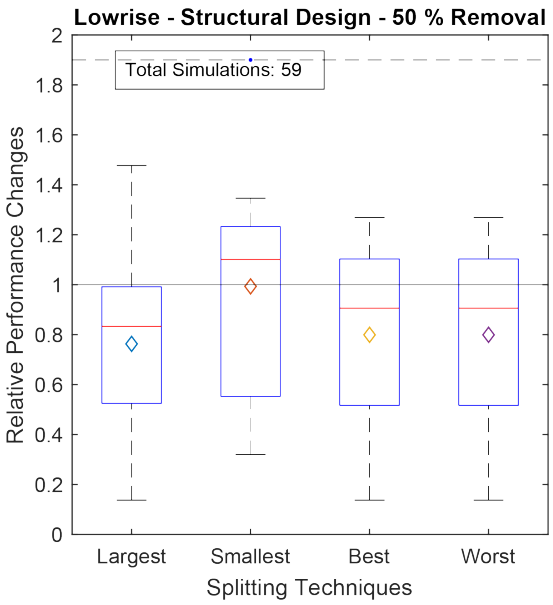


(c) Highrise Design

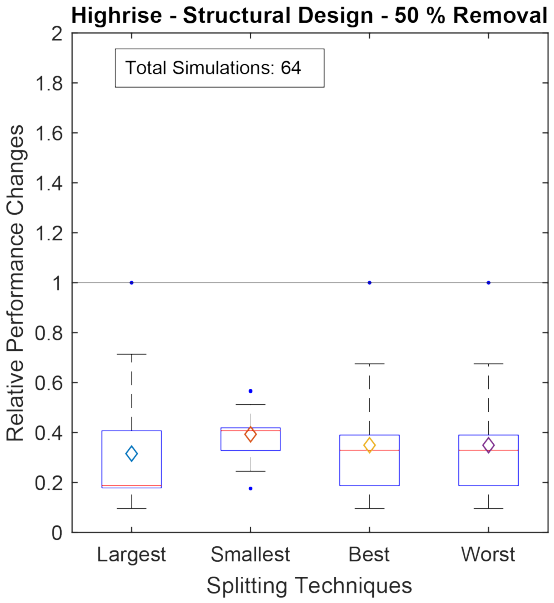
Figure B.4: Normalized simulation results from set 1 for three different building spatial designs. Normalized values are calculated towards the performance of the initial design. For each design the difference between splitting different spaces is shown.



(a) Cube Design

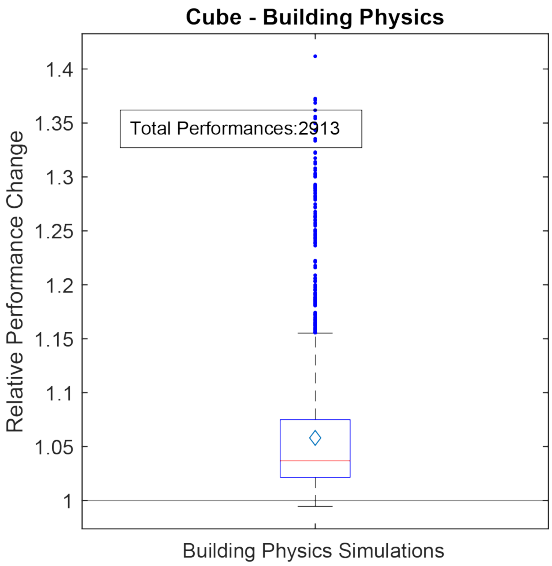


(b) Lowrise Design

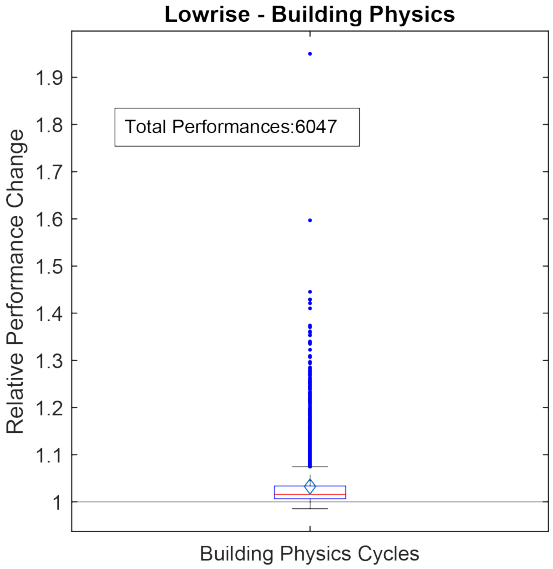


(c) Highrise Design

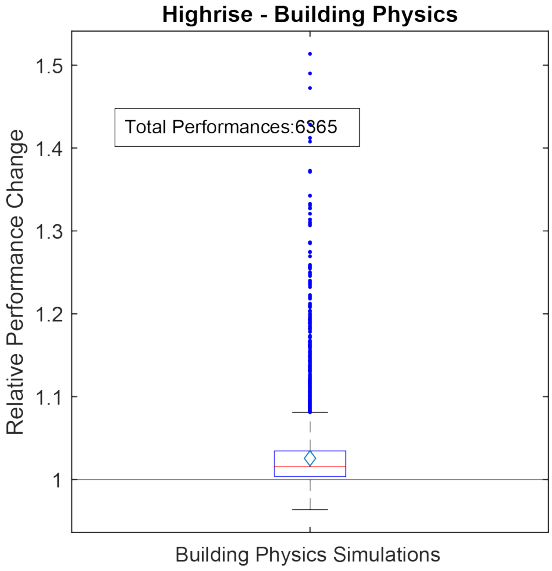
Figure B.5: Normalized simulation results for set 1 for three different building spatial designs. Normalized values are calculated towards the performance of the initial design. For each design the difference between splitting different spaces is shown. Only solutions are shown which have the removal parameter of 50% removed.



(a) Cube Design



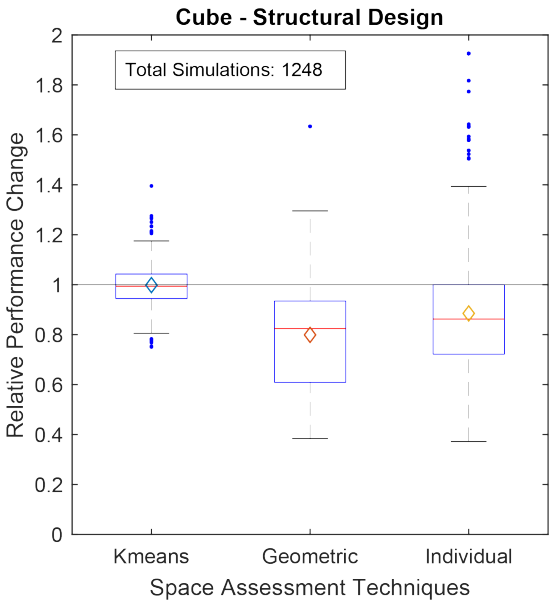
(b) Lowrise Design



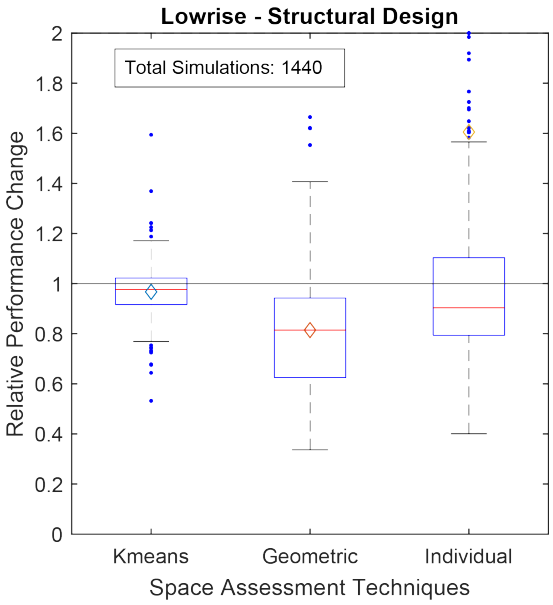
(c) Highrise Design

Figure B.6: Normalized simulation results for set 1 for three different building spatial designs for building physics. Normalized values are calculated towards the performance of the initial design.

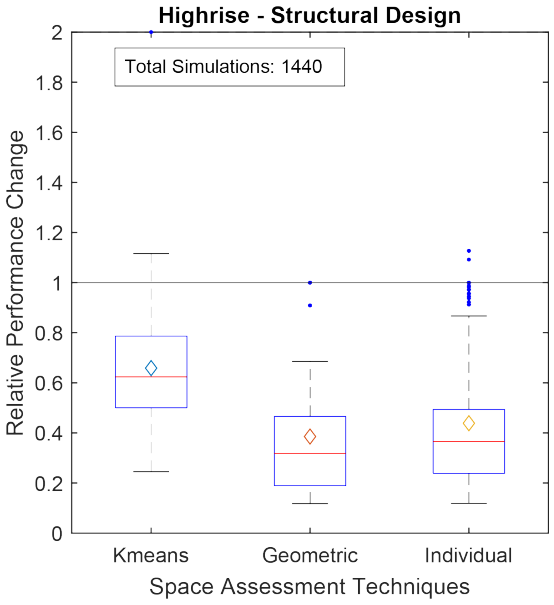
B.2 Parameter set 2



(a) Cube Design



(b) Lowrise Design



(c) Highrise Design

Figure B.7: Performance assessment results for set 2 for structural performances.

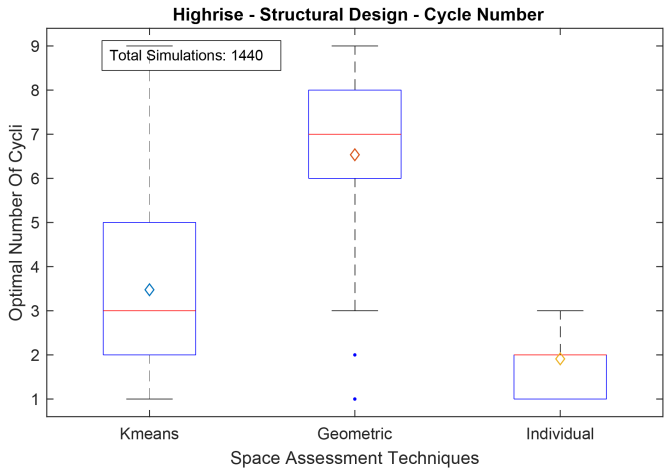


Figure B.8: Optimal amount of cycles for the optimization of the highrise design towards structural performance for different assessment techniques for set 2.

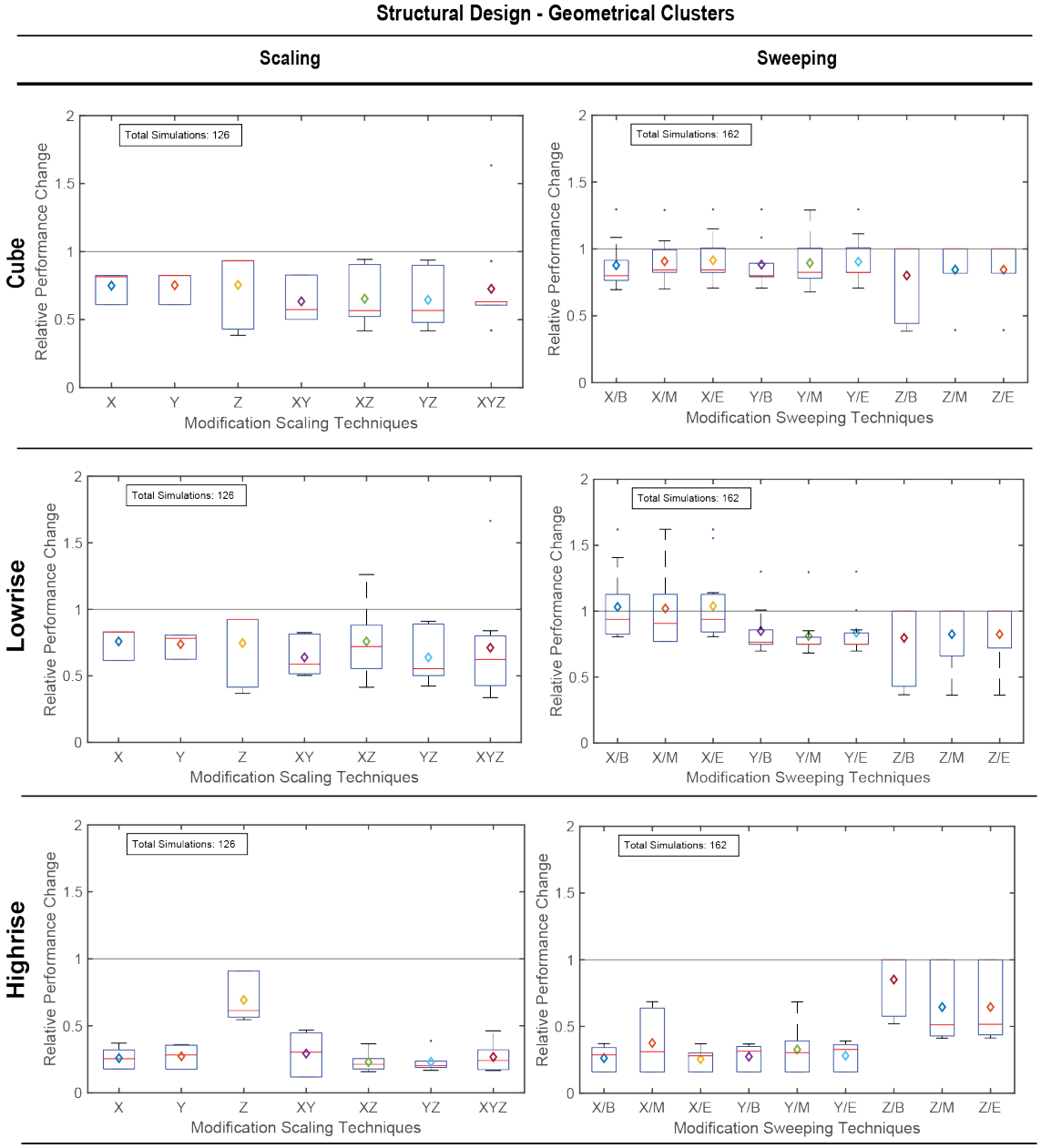
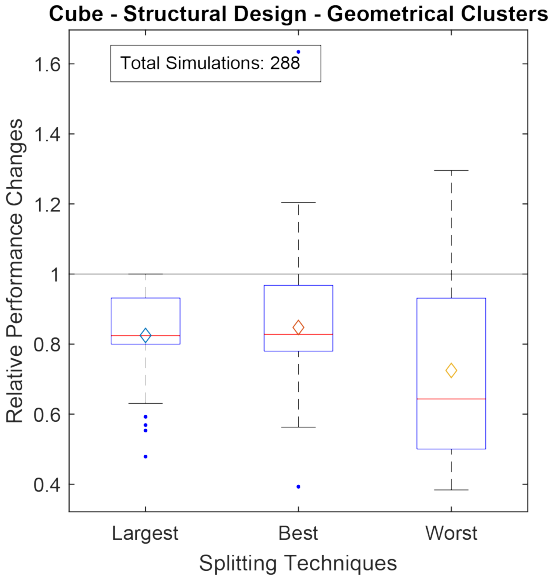
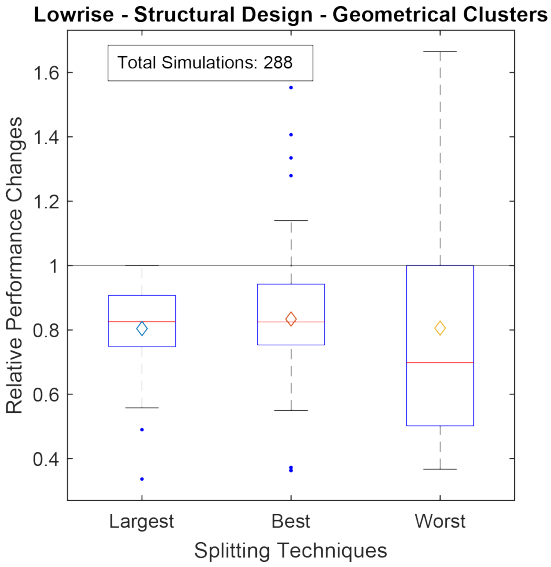


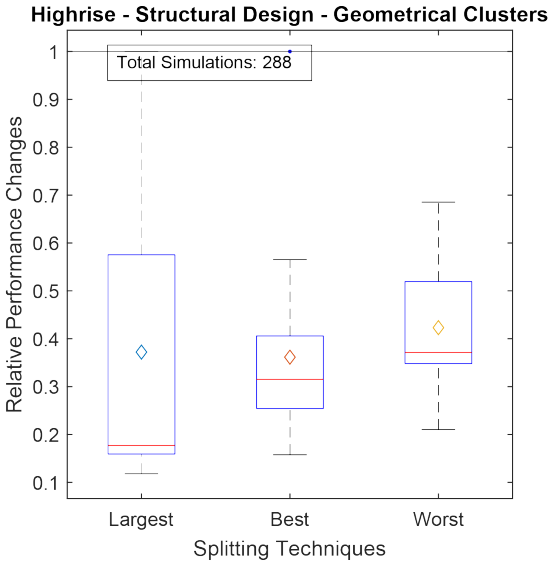
Figure B.9: Scaling and sweeping modification results for structural performances for set 2 for the geometrical cluster technique. For scaling the horizontal axis describes over which axis the building spatial design is scaled. The horizontal axis for sweeping consist out of two elements: the first describes over which axis the sweep is executed, the second at which location is the building spatial design with B, M, and E for begin, middle, and end respectively.



(a) Cube Design

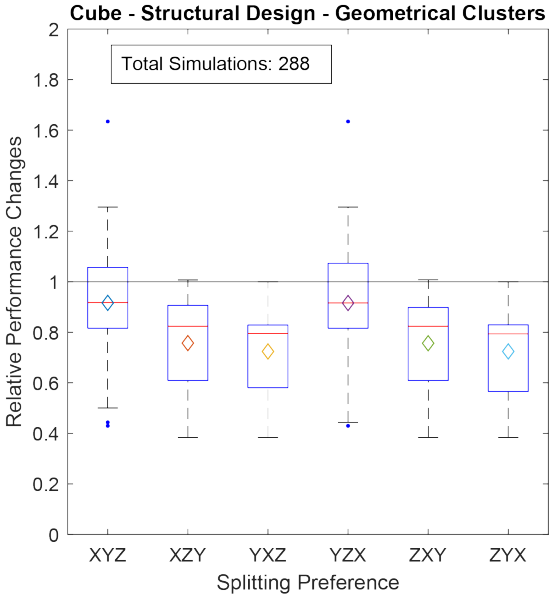


(b) Lowrise Design

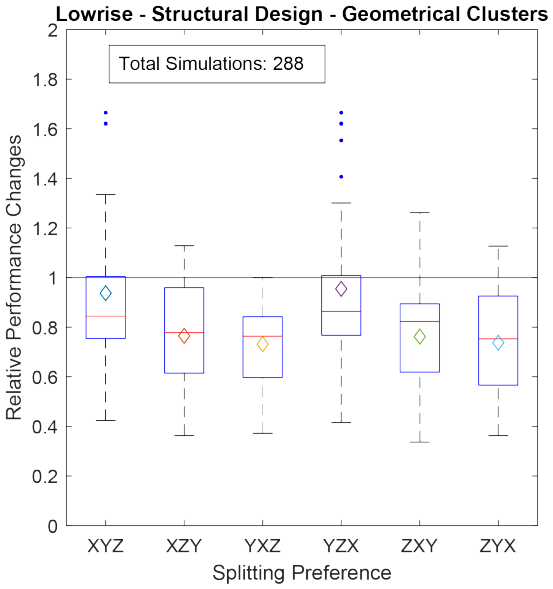


(c) Highrise Design

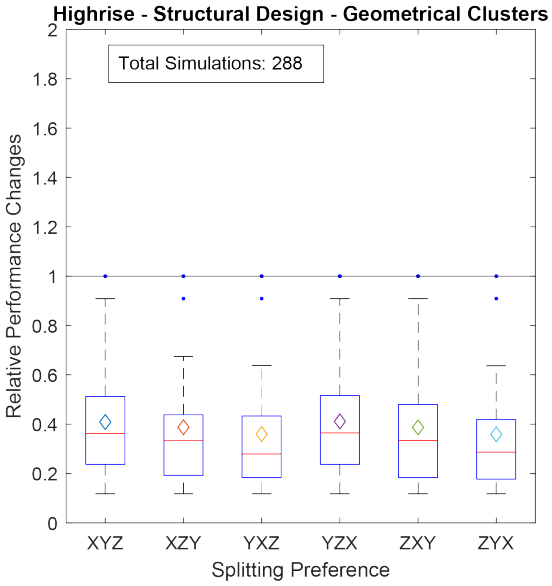
Figure B.10: Results for different split techniques for parameter set 2 for structural performances.



(a) Cube Design

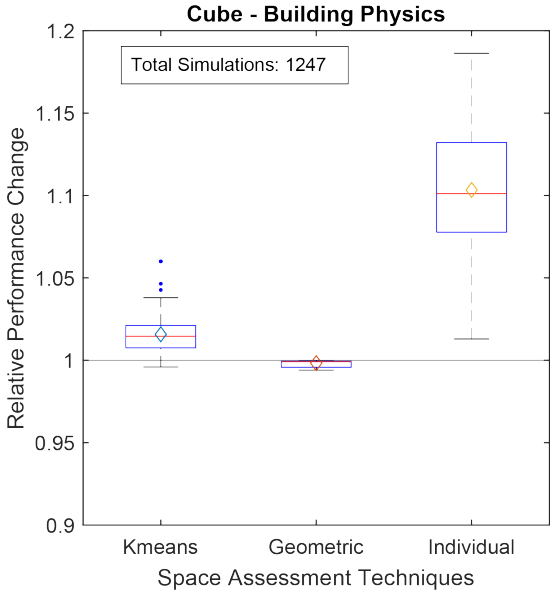


(b) Lowrise Design

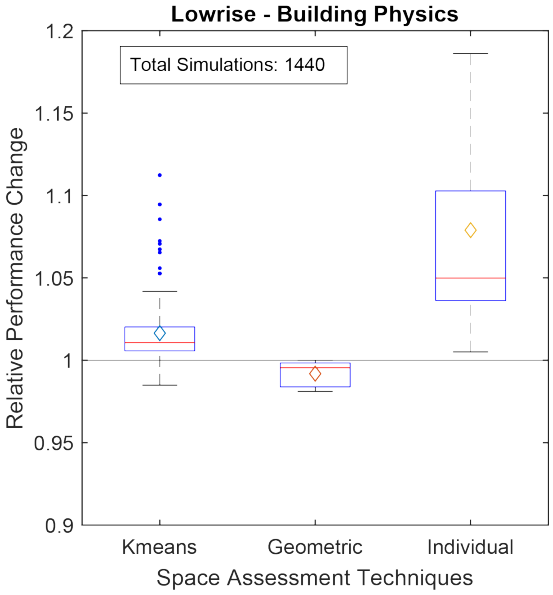


(c) Highrise Design

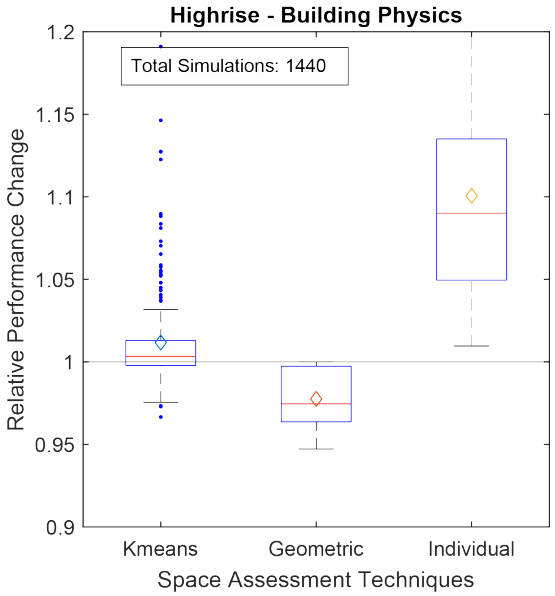
Figure B.11: Results for different split techniques for parameter set 2 for structural performances.



(a) Cube Design



(b) Lowrise Design



(c) Highrise Design

Figure B.12: Performance assessment results for set 2 for building physics performances.

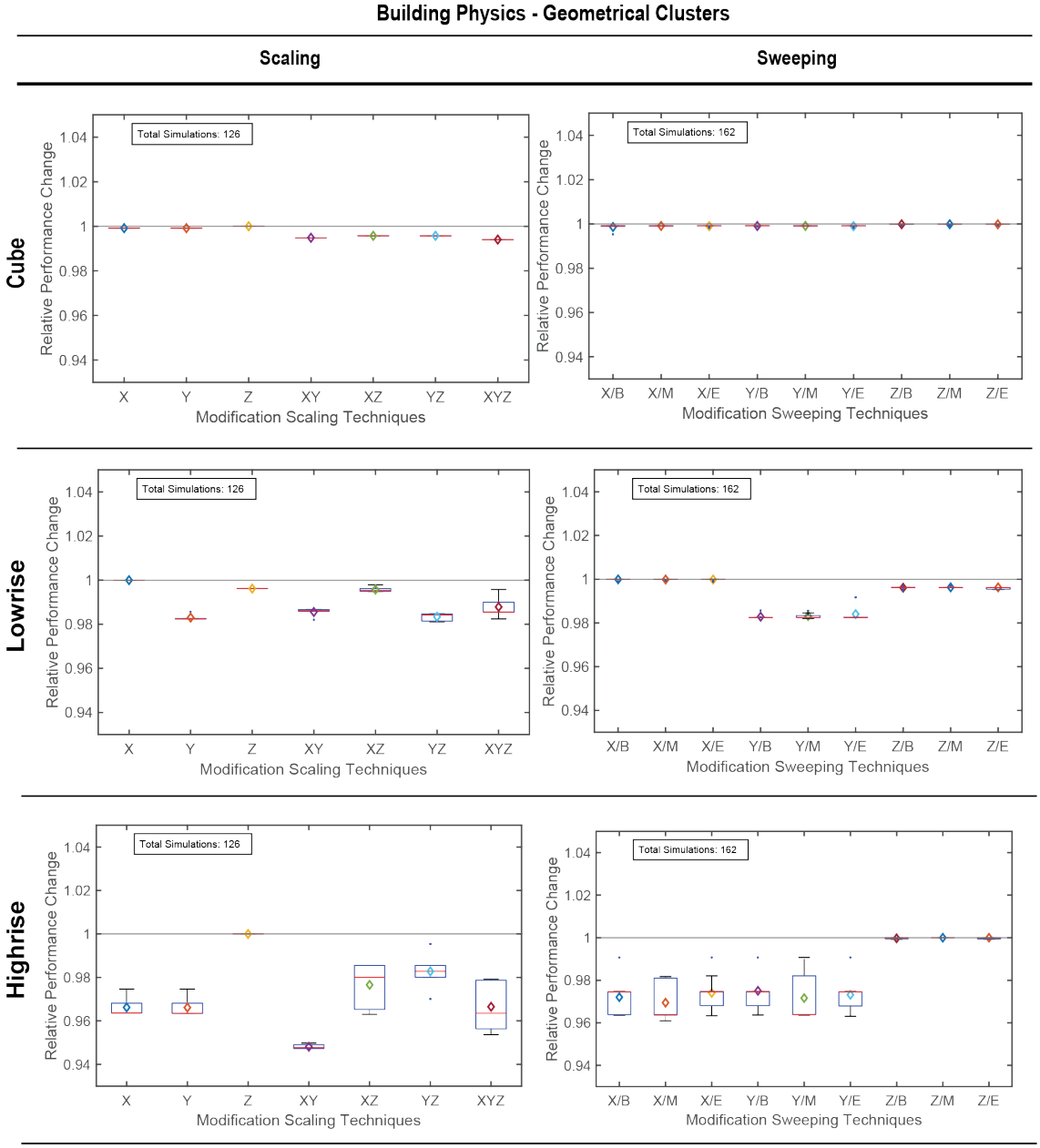
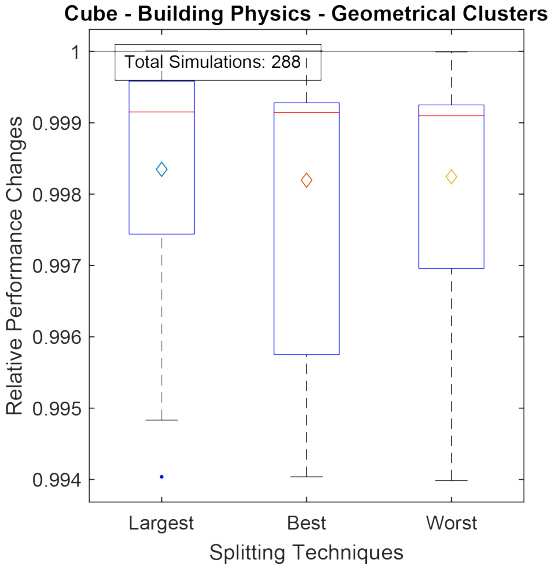
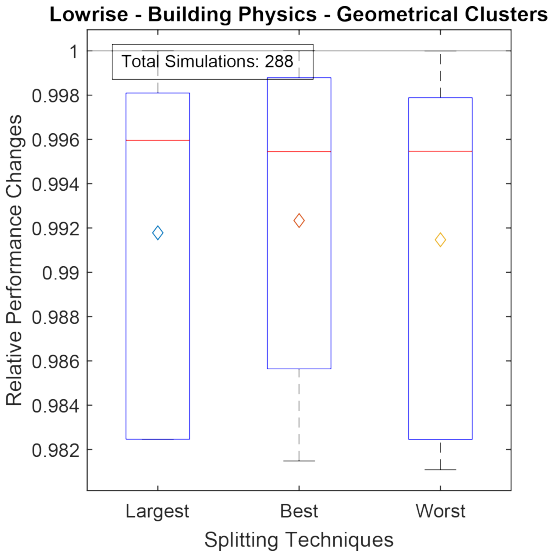


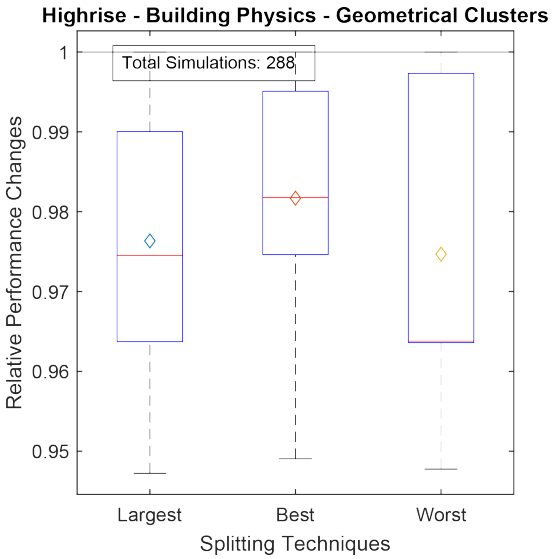
Figure B.13: Scaling and sweeping modification results for building physics performances for set 2 for the geometrical cluster technique. For scaling the horizontal axis describes over which axis the building spatial design is scaled. The horizontal axis for sweeping consist out of two elements: the first describes over which axis the sweep is executed, the second at which location is the building spatial design with B, M, and E for begin, middle, and end respectively.



(a) Cube Design

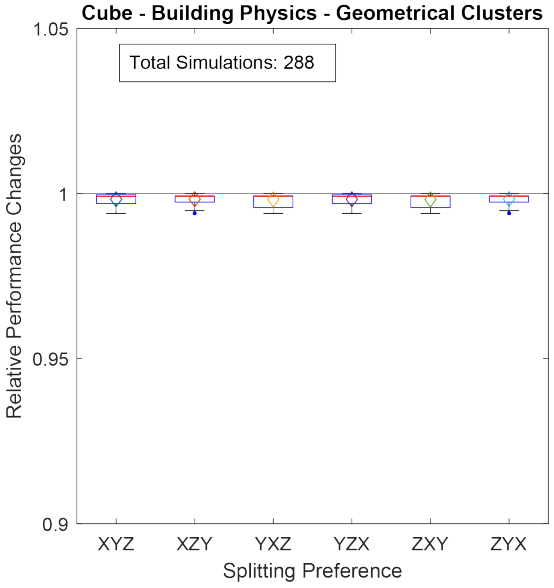


(b) Lowrise Design

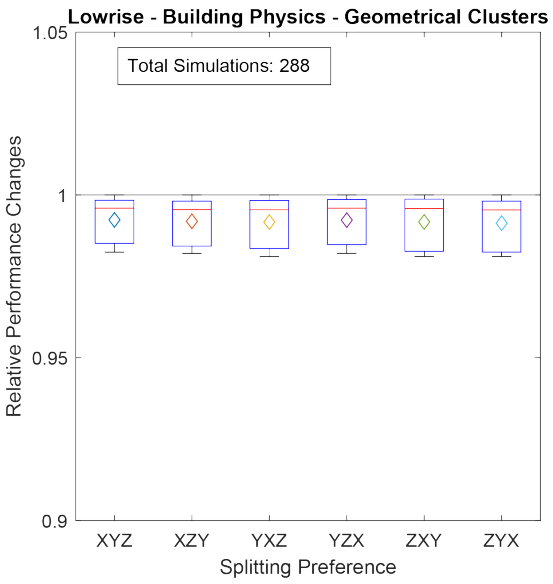


(c) Highrise Design

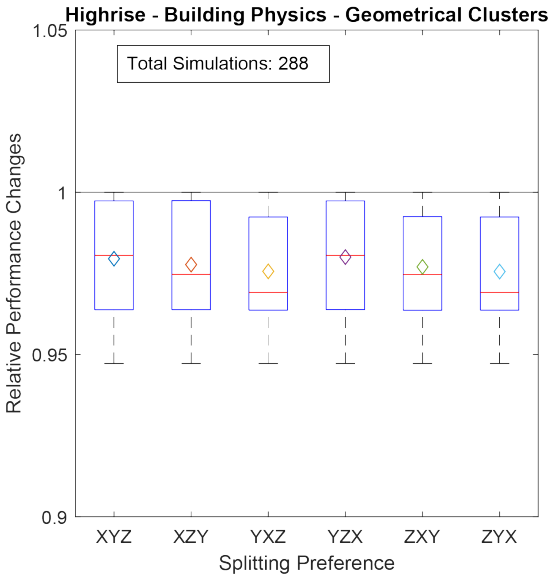
Figure B.14: Results for different split techniques for parameter set 2 for building physics optimization.



(a) Cube Design



(b) Lowrise Design



(c) Highrise Design

Figure B.15: Results for different split techniques for parameter set 2 for building physics performances.

C Code

This appendix contains the C++ code as developed in this research. First the main file is presented which contains the logic for the loop and optimization. After which the HBO settings, Building Performances, Performance Evaluation, Non Aggregate Performances, Aggregate Performances, Space Ranking, Building Modification, Space Removal, Rescaling, Sweeping, Splitting, and Non Feasible Solutions are provided. The final piece of code given is the Movable Sizable document, this provides the movable sizable representation as used in this research. However, only the last bit of code has been developed in this research, the major part is developed by S. Boonstra and is presented in the multiple of his researches (2016, 2018).

The C++ code files included in order of appearance:
main.cpp
HBO_Settings.hpp
HBO_Settings.cpp
Building_Performance.hpp
Building_Performances.cpp
Performance_Evaluation.hpp
Non˙Aggregate_Performances.hpp
Aggregate_Performances.hpp
Space_Ranking.hpp
Building_Modification.hpp
Space_Removal.hpp
Rescaling.hpp
Sweeping.hpp
Splitting.hpp
Non_Feasible_Solutions.hpp
Movable_Sizable.hpp

```cpp
#include <BSO/Structural_Design/SD_Analysis.hpp>
#include <BSO/Building_Physics/BP_Simulation.hpp>
#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/Visualisation/Visualisation.hpp>
#include <BSO/Performance_Indexing.hpp>

#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/HBO_Settings.hpp>

#include <BSO/HBO/Performance_Evaluation/Building_Performances.cpp>
#include <BSO/HBO/HBO_Settings.cpp>

#include <BSO/HBO/Performance_Evaluation/Performance_Evaluation.hpp>
#include <BSO/HBO/Space_Selection/Space_Ranking.hpp>
#include <BSO/HBO/Space_Selection/Space_Removal.hpp>
#include <BSO/HBO/Building_Modification/Building_Modification.hpp>
#include <BSO/HBO/Building_Modification/Splitting.hpp>
#include <BSO/HBO/Building_Modification/Rescaling.hpp>
#include <BSO/HBO/Non_Feasible_Solutions.hpp>

#include <BSO/Trim_And_Cast.hpp>

#include <AEI Grammar/Grammar_2.hpp>

#include <iostream>
#include <vector>
#include <ctime>
#include <string>
#include <fstream>

/* Parameter_Function */

BSO::Structural_Design::SD_Building_Results Calculate_SD(BSO::Spatial_Design::MS_Building& Building );
BSO::Building_Physics::BP_Building_Results Calculate_BP(BSO::Spatial_Design::MS_Building& Building );

int main(int argc, char* argv[])
{
    std::vector<std::string> args( argv+1, argv+argc ) ;

    // settings for this study
    BSO::HBO::Settings settings ;
    bool visualise = false;

    std::string name ;
    std::string input_file ;
    std::string output_file ;
    std::string output_file_results ;
    std::string output_file_solutions ;

    int amount removed ;
    int cycles = 0 ;
    int snap = 10 ;

    bool sd sim = true ;
    bool bp sim = true ;

    // initiate the log files in which the progress during simulation can be tracked
    std::ofstream log;
    log.open("log.txt", std::ofstream::out | std::ofstream::trunc);
    log.close();

    auto it = args.begin();
    if (it == args.end())
    {
        std::cerr << "Error, expected arguments, got nothing, -h or --help for help, exiting now..." << std::endl;
        exit(1);
    }
    while (it != args.end())
    {
        if (*it == "-h" || *it == "--help")
        {
            std::cout << "\n\nThis program is made to do a parameter study for the sweeping building modification"
                      << "\nOptions are:" << std::endl;

            std::cout << "-h\t--help\t\t"       << "Will give you this help menu.\n"
                      << std::endl;

            std::cout << "-i\t--input\t\t"      << "Allows you to specify the input file.\n"
                      << std::endl;

```

```
81              std::cout << "-d\t--discipline\t\t" << "Allows you to specify the disciplines used
                in optimization. \n"
82                        << std::endl;
83
84              std::cout << "-cs\t--compliance_sd\t\t" << "Determines whether a high or low
                compliance for SD is considered best or worst. \n "
85                        << std::endl;
86
87              std::cout << "-v\t--visualise\t"    << "Will give you a visualisation of the input.\n"
88                        << std::endl;
89
90              std::cout << "-c\t--cycles\t"       << "Specifies the number of simulation cycles."
91                        << "\n\t\t\tExpects a positive integer greater than 0.\n"
92                        << std::endl;
93
94              std::cout << "-al\t--assessment_level\t"        << "Allows you to specify whether
                the optimization is done using clusters or individual spaces.\n"
95                        << std::endl;
96
97              std::cout << "-pa\t--performance_assessment\t"  << "Determines whether an
                aggregation is used or not.\n"
98                        << std::endl;
99
100             std::cout << "-ag\t--aggregate\t"   << "Specifies the aggregation method used.\n"
101                       << std::endl;
102
103             std::cout << "-sr\t--space_ranking\t"           << "Specifies the approach of
                creating rankings of the spaces.\n"
104                       << std::endl;
105
106             std::cout << "-rt\t--removal_type\t" << "Specifies the approach of space removal.\n"
107                       << std::endl;
108
109             std::cout << "-ar\t--amount_removed\t" << "Allows for the specification of the
                amount of spaces to be removed.\n"
110                       << std::endl;
111
112             std::cout << "-m\t--modification\t" << "Specifies the modification approach.\n"
113                       << std::endl;
114
115             std::cout << "-sc\t--scaling\t"     << "Specifies over which axis the scaling
                function operates.\n"
116                       << std::endl;
117
118             std::cout << "-sw\t--sweeping\t"    << "Specifies which sweeping option is used.\n"
119                       << std::endl;
120
121             std::cout << "-sp\t--splitting_option\t"        << "Specifies which types of spaces
                are split.\n"
122                       << std::endl;
123             std::cout << "-pr\t--splitting preference\t"    << "Specifies the preference in
                which spaces are split. \n"
124                       << std::endl;
125             return 0; // end the program
126         }
127
128         else if (*it == "-i" || *it == "--input")
129         {
130             if (++it == args.end() || (*it)[0] == '-')
131                 throw std::domain_error("Expected a value after -i or --input");
132
133             name = *it;
134             boost::trim(name);
135
136             it++;
137         }
138         else if ( *it == "-d" || *it == "--discipline" )
139         {
140             if ( ++it == args.end() || (*it)[0] == '-' )
141                 throw std::domain_error( "Expected a value after -d or --discipline" ) ;
142             int disc = BSO::trim_and_cast_uint( *it ) ;
143
144             if ( disc >= (unsigned int)BSO::HBO::discipline::ARG_COUNT )
145                 throw std::domain_error( "Discipline value is larger than ARG_COUNT" ) ;
146
147             settings.discipline_involved = BSO::HBO::discipline( disc ) ;
148
149             it++ ;
150         }
151         else if ( *it == "-cs" || *it == "--compliance_sd" ) // if called the program
152         {
153             if ( ++it == args.end() || (*it)[0] == '-' )
154                 throw std::domain_error( "Expected a value after -cs or --compliance_sd" ) ;
155             int cs = BSO::trim_and_cast_uint( *it ) ;
```

```
156
157             if ( cs == 1 )
158                 settings.inverse_SD = true ;
159             else if ( cs == 0 )
160                 settings.inverse_SD = false ;
161             else
162                 throw std::domain_error( "Expected a 0 or 1 after -cs or --compliance_sd");
163
164             it++ ;
165         }
166         else if (*it == "-v" || *it == "--visualise")
167         {
168             visualise = true;
169
170             it++;
171         }
172         else if (*it == "-c" || *it == "--cycles")
173         {
174             if (++it == args.end() || (*it)[0] == '-')
175                 throw std::domain_error("Expected a value after -c or --cycles");
176
177             int temp_cycle = BSO::trim_and_cast_uint( *it ) ;
178             cycles = temp_cycle ;
179
180             if (cycles < 1)
181                 throw std::domain_error("Expected a positive int greater than zero after -c or
                    --cycles");
182
183             it++;
184         }
185
186         else if ( *it == "-al" || *it == "--assessment_level" )
187         {
188             if ( ++it == args.end() || (*it)[0] == '-' )
189                 throw std::domain_error( "Expected a value after -al or --assessment_level" ) ;
190
191             int ass_lvl = BSO::trim_and_cast_uint( *it ) ;
192
193             if ( ass_lvl >= (unsigned int)BSO::HBO::assessment_level::ARG_COUNT )
194                 throw std::domain_error( "Error, assessment level argument is larger then
                    ARG_COUNT" ) ;
195
196             settings.individual_or_clus = BSO::HBO::assessment_level( ass_lvl ) ;
197
198             ++it;
199         }
200         else if ( *it == "-pa" || *it == "performance_assessment" )
201         {
202             if ( ++it == args.end() || (*it)[0] == '-' )
203                 throw std::domain_error("Expected a value after -pa or --performance_assessment"
                    ) ;
204
205             int perf ass = BSO::trim and cast uint( *it ) ;
206
207             if (perf ass >= (unsigned int)BSO::HBO::performance assessment::ARG COUNT )
208                 throw std::domain error("Error, performance assessment argument is larger then
                    ARG COUNT" ) ;
209
210             settings.assessment_options = BSO::HBO::performance_assessment( perf_ass ) ;
211             it++ ;
212         }
213         else if ( *it == "-ag" || *it == "--aggregate" )
214         {
215             if ( ++it == args.end() || (*it)[0] == '-' )
216                 throw std::domain_error( "Expected a value after -ag or --aggregate" ) ;
217             int agg = BSO::trim_and_cast_uint( *it ) ;
218
219             if ( agg >= (unsigned int)BSO::HBO::aggregate_disciplines::ARG_COUNT )
220                 throw std::domain_error("Error, aggregate options is larger then ARG_COUNT" ) ;
221             settings.aggregate_options = BSO::HBO::aggregate_disciplines(agg);
222
223             it++ ;
224         }
225         else if ( *it == "-sr" || *it == "--space_ranking" )
226         {
227             if ( ++it == args.end() || (*it)[0] == '-' )
228                 throw std::domain_error("Expected a value after -sr or --space_ranking" ) ;
229             int space_rank = BSO::trim_and_cast_uint( *it ) ;
230
231             if ( space_rank >= (unsigned int)BSO::HBO::space_ranking::ARG_COUNT )
232                 throw std::domain_error("Error, space rankinig is larger then ARG_COUNT" ) ;
233             settings.ranking_type = BSO::HBO::space_ranking(space_rank);
234
235             it++ ;
```

```
236             }
237             else if ( *it == "-rt" || *it == "--removal_type" )
238             {
239                 if ( ++it == args.end() || (*it)[0] == '-' )
240                     throw std::domain_error( "Expected a value after -rt or --removal_type" ) ;
241                 int remove_type = BSO::trim_and_cast_uint( *it );
242 
243                 if ( remove_type >= (unsigned int)BSO::HBO::space_removal_type::ARG_COUNT )
244                     throw std::domain_error( "Error, removal type argument is larger then ARG_COUNT"
                        ) ;
245                 settings.removal_type = BSO::HBO::space_removal_type( remove_type ) ;
246                 it++;
247             }
248             else if ( *it == "-ar" || *it == "--amount_removed" )
249             {
250                 if ( ++it == args.end() || (*it)[0] == '-' )
251                     throw std::domain_error( "Expected a value after -ar or --amount_removed" ) ;
252                 int remove = BSO::trim_and_cast_uint ( *it ) ;
253 
254                 amount_removed = remove ;
255                 it++ ;
256             }
257             else if ( *it == "-m" || *it == "--modification" )
258             {
259                 if ( ++it == args.end() || (*it)[0] == '-' )
260                     throw std::domain_error("Expected a value after -m or --modification") ;
261 
262                 int mod = BSO::trim_and_cast_uint( *it ) ;
263 
264                 if ( mod >= (unsigned int)BSO::HBO::building_modification::ARG_COUNT )
265                     throw std::domain_error( "Error, modification argument is larger then ARG_COUNT"
                        ) ;
266 
267                 settings.modification_options = BSO::HBO::building_modification( mod ) ;
268 
269                 ++it ;
270             }
271             else if ( *it == "-sc" || *it == "--scaling" )
272             {
273                 if ( ++it == args.end() || (*it)[0] == '-' )
274                     throw std::domain_error( "Expected a value after -sc or --scaling" ) ;
275                 int scale = BSO::trim_and_cast_uint( *it ) ;
276 
277                 if ( scale >= (unsigned int)BSO::HBO::rescaling_options::ARG_COUNT )
278                     throw std::domain_error( "Error, rescaling argument larger then ARG_COUNT" ) ;
279                 settings.rescaling = BSO::HBO::rescaling_options( scale ) ;
280                 it++;
281             }
282             else if ( *it == "-sw" || *it == "--sweeping" )
283             {
284                 if ( ++it == args.end() || (*it)[0] == '-' )
285                     throw std::domain_error("Expected a valua after -sw or --sweeping") ;
286                 int sweep = BSO::trim and cast uint( *it ) ;
287 
288                 if ( sweep >= (unsigned int)BSO::HBO::sweeping options::ARG COUNT )
289                     throw std::domain error( "Error, sweeping arguments larger than ARG COUNT" ) ;
290                 settings.sweep option = BSO::HBO::sweeping options( sweep );
291                 it++ ;
292             }
293             else if ( *it == "-sp" || *it == "--splitting_option" )
294             {
295                 if ( ++it == args.end() || (*it)[0] == '-' )
296                     throw std::domain_error("Expected a value after -sp or --splitting_option" ) ;
297                 int split = BSO::trim_and_cast_uint( *it ) ;
298 
299                 if ( split >= (unsigned int)BSO::HBO::splitting_options::ARG_COUNT )
300                     throw std::domain_error( "Error, splitting option argument larger than
                        ARG_COUNT" ) ;
301                 settings.split = BSO::HBO::splitting_options( split ) ;
302 
303                 it++ ;
304             }
305             else if ( *it == "-pr" || *it == "--splitting_preference" )
306             {
307                 if ( ++it == args.end() || (*it)[0] == '-' )
308                     throw std::domain_error( "Expected a value after -pr or --splitting_preference"
                        ) ;
309 
310                 int split_prefence = BSO::trim_and_cast_uint( *it ) ;
311 
312                 if ( split_prefence >= (unsigned int) BSO::HBO::splitting_preference::ARG_COUNT )
313                     throw std::domain_error( "Error, splitting preference argument larger than
                        ARG_COUNT" ) ;
314 
```

```
315             settings.split_pref = BSO::HBO::splitting_preference( split_prefence );
316
317             it++ ;
318         }
319         else if ( *it == "-cl" || *it == "--cluster_type" )
320         {
321             if ( ++it == args.end() || (*it)[0] == '-' )
322                 throw std::domain_error( "Expected a value after -cl or --cluster_type" ) ;
323             int cluster = BSO::trim_and_cast_uint( *it ) ;
324
325             if ( cluster > ( unsigned int ) BSO::HBO::cluster_type::ARG_COUNT )
326                 throw std::domain_error( "Error, cluster argument larger than ARG_COUNT " );
327             settings.clus_type = BSO::HBO::cluster_type( cluster ) ;
328
329             it++ ;
330         }
331         else
332         {
333             throw std::invalid_argument("Argument: " + *it + " not recognised");
334         }
335     }
336
337
338     input_file = "/home/tomas/Documents/1_Parameterstudy/0_Input/" + name  + ".txt";
339     output_file_results = "/home/tomas/Documents/1_Parameterstudy/0_Output/0_Result_Files/" +
        name + "_Test_Results.txt" ;
340     output_file_solutions = "/home/tomas/Documents/1_Parameterstudy/0_Output/" + name + "/" +
        name ;
341
342     BSO::Spatial_Design::MS_Building initial_design(input_file);
343
344     // Results vectors
345     std::vector<BSO::Spatial_Design::MS_Building> building_designs ;
346     std::vector<BSO::Spatial_Design::MS_Building> non_feasible_results ;
347
348     // Calculation vectors
349     std::vector<BSO::Spatial_Design::MS_Building> building_calc(2) ;
350     std::vector<BSO::HBO::Performance_Evaluation::Building_Performances> build_perform(2) ;
351
352
353     if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
        settings.discipline_involved == BSO::HBO::discipline::SD )
354         settings.sd_results = Calculate_SD( initial_design ) ;
355     if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
        settings.discipline_involved == BSO::HBO::discipline::BP )
356         settings.bp_results = Calculate_BP( initial_design ) ;
357
358     building_calc[ 0 ] = initial_design ;
359     BSO::HBO::Performance_Evaluation::Building_Performances initial_build_perform(
        initial_design, settings ) ;
360     build_perform[ 0 ] = initial_build_perform ;
361
362     settings.set_cycles( initial_build_perform ) ;
363     building_calc.resize( settings.cycles + 51 ) ;
364     build_perform.resize( settings.cycles + 51 ) ;
365
366     std::cout   << "SETTINGS" << std::endl ;
367     settings.cout_file() ;
368     std::cout   << "BUILDING RESULTS" << std::endl
369                 << "ITERATION " ;
370
371     if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
        settings.discipline_involved == BSO::HBO::discipline::SD )
372         std::cout << "SD COMPLIANCE, SPACE COUNT, BUILDING VOLUME, REALISTIC " ;
373
374     if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
        settings.discipline_involved == BSO::HBO::discipline::BP )
375         std::cout << "BP COMPLIANCE, SPACE COUNT, BUILDING VOLUME,  REALISTIC" ;
376
377     std::cout << std::endl
378         << 0 ;
379
380     if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
        settings.discipline_involved == BSO::HBO::discipline::SD )
381         std::cout << " " << settings.sd_results.m_total_compliance << " " << building_calc[ 0
            ].obtain_space_count() << " " << building_calc[ 0 ].get_volume() << " " <<
            settings.realistic_building ;
382
383     if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
        settings.discipline_involved == BSO::HBO::discipline::BP )
384         std::cout << " " << settings.bp_results.m_total_energy << " " << building_calc[ 0
            ].obtain_space_count() << " " << building_calc[ 0 ].get_volume() << " " <<
            settings.realistic_building ;
385
```

```
386     std::cout << std::endl ;
387
388     for ( unsigned int i = 1 ; i <= settings.cycles ; i++ )
389     {
390         build_perform[ i - 1 ] = BSO::HBO::Performance_Evaluation::evaluate_performance(
            build_perform[ i - 1 ], settings ) ;
391         if ( settings.terminate_simulation ) { break ; }
392         std::vector<int> ranked_spaces = BSO::HBO::Space_Selection::rank_spaces( build_perform[
            i - 1 ], settings ) ;
393         if ( settings.terminate_simulation ) { break ; }
394         building_calc[ i - 1 ] = BSO::HBO::Space_Selection::removal_of_spaces( building_calc[ i
            - 1 ], ranked_spaces, build_perform[ i -1 ], settings );
395         if ( settings.terminate_simulation ) { break ; }
396         building_calc[ i ] = BSO::HBO::Building_Modification::building_modification(
            building_calc[ i - 1 ], build_perform[ i - 1 ], settings) ;
397         if ( settings.terminate_simulation ) { break ; }
398         building_calc[ i ] = BSO::HBO::Building_Modification::split_spaces( building_calc[ i ],
            build_perform[ i - 1 ], settings) ;
399         if ( settings.terminate_simulation ) { break ; }
400         building_calc[ i ].reset_z_zero() ;
401
402         building_calc[ i ].snap_on( 10 ) ;
403
404         // write an .txt output file with the nwe building spatial design
405         int j = (unsigned int)settings.discipline_involved ;
406
407         std::string solution_name = output_file_solutions + "_" + std::to_string( (unsigned int)
            settings.discipline_involved ) + std::to_string( (unsigned int)
            settings.individual_or_clus ) ;
408
409         if ( settings.individual_or_clus == BSO::HBO::assessment_level::CLUSTERS )
410         {
411             solution_name += std::to_string( (unsigned int) settings.clus_type ) ;
412         }
413         else if ( settings.individual_or_clus == BSO::HBO::assessment_level::INDIVIDUAL )
414         {
415             // do nothing
416         }
417         solution_name += std::to_string( (unsigned int) settings.assessment_options ) +
            std::to_string( (unsigned int) settings.ranking_type ) + std::to_string( (unsigned int)
            settings.removal_type ) + std::to_string( (unsigned int) settings.modification_options ) ;
418         if ( settings.modification_options == BSO::HBO::building_modification::SCALE )
419             solution_name += std::to_string( (unsigned int) settings.rescaling ) ;
420         else if ( settings.modification_options == BSO::HBO::building_modification::SWEEP )
421             solution_name += std::to_string( (unsigned int) settings.sweep_option ) ;
422         else { std::cout << "Error in solution_name building_modification, exiting now ... (
            main.cpp) " << std::endl; exit(1) ; }
423
424         solution_name += std::to_string( (unsigned int) settings.split ) + std::to_string(
            (unsigned int) settings.split_pref ) + "_" + std::to_string( i ) + ".txt" ;
425
426         if ( settings.terminate simulation ) { break ; }
427         building calc[ i ].write file( solution name ) ;
428
429         // check if the building spatial design is feasible
430         if ( BSO::HBO::detect infinity(building calc[ i ] ) )
431         {
432             settings.terminate simulation = true ;
433             // if not feasible store in a different vector
434             non_feasible_results.push_back( building_calc[ i ] ) ;
435             break ;
436         }
437         else if ( BSO::HBO::detect_floating_space( building_calc[ i ] ) )
438         {
439             settings.terminate_simulation = true ;
440             // if not feasible store in a different vector
441             non_feasible_results.push_back( building_calc[ i ] ) ;
442             break ;
443         }
444         else
445         {
446             // if feasible store in the general solutions vector
447             building_designs.push_back( building_calc[ i ] ) ;
448             space_boundary_conditions_check( building_calc[ i ], settings ) ;
449
450             if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
                settings.discipline_involved == BSO::HBO::discipline::SD )
451                 settings.sd_results = Calculate_SD( building_calc[ i ] ) ;
452             if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
                settings.discipline_involved == BSO::HBO::discipline::BP )
453                 settings.bp_results = Calculate_BP( building_calc[ i ] ) ;
454
455             BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform(
                building_calc[ i ], settings ) ;
```

```
456             build_perform[ i ] = temp_build_perform ;
457
458             std::cout << i << " " ;
459             if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
                settings.discipline_involved == BSO::HBO::discipline::SD )
460                 std::cout << settings.sd_results.m_total_compliance << " " << building_calc[ i
                    ].obtain_space_count() << " " << building_calc[ i ].get_volume() << " " <<
                    settings.realistic_building ;
461             if ( settings.discipline_involved == BSO::HBO::discipline::ALL ||
                settings.discipline_involved == BSO::HBO::discipline::BP )
462                 std::cout << settings.bp_results.m_total_energy << " " << building_calc[ i
                    ].obtain_space_count() << " " << building_calc[ i ].get_volume() << " " <<
                    settings.realistic_building;
463             std::cout << std::endl ;
464         }
465     }
466
467     if ( settings.terminate_simulation )
468     {
469         std::cout << "Simulation terminated prematurely" <<std::endl ;
470     }
471     else if ( !settings.terminate_simulation )
472     {
473         std::cout << "Simulation completed" << std::endl ;
474     }
475
476     if ( visualise ) // if a visualisation is requested
477     {
478         BSO::Visualisation::init_visualisation( argc, argv ) ;
479
480         BSO::Spatial_Design::MS_Conformal CF(building_designs.back(), &(BSO::Grammar::grammar_2)
            );
481
482         BSO::Visualisation::visualise( initial_design ) ;
483
484         for ( unsigned int i = 0 ; i < building_designs.size() ; i++ )
485         {
486             BSO::Visualisation::visualise( building_designs[i] );
487         }
488
489         for ( unsigned int i = 0 ; i < non_feasible_results.size() ; i++ )
490         {
491             if( !BSO::HBO::detect_infinity( non_feasible_results[i] ) ) // solutions with
                infinite numbers cannot be visualised due to their nature
492                 BSO::Visualisation::visualise( non_feasible_results[i] ) ;
493         }
494
495         BSO::Visualisation::end_visualisation() ;
496     }
497
498     return 0;
499 }
500
501 BSO::Structural Design::SD Building Results Calculate SD(BSO::Spatial Design::MS Building&
    Building )
502 {
503     BSO::Structural Design::SD Building Results temp result ;
504     static int SD calc number = 0 ; SD calc number++ ;
505
506     BSO::Spatial_Design::MS_Conformal CF (Building, &(BSO::Grammar::grammar_2) );
507
508     BSO::Structural_Design::SD_Analysis SD_building(CF); //std::cout << "Created the Structural
        Design. - " << std::endl;
509
510     SD_building.analyse() ;
511     temp_result = SD_building.get_results() ; //std::cout << "Analysed the structural design,
        total compliance: " << temp_result.m_total_compliance << std::endl ;
512
513     BSO::SD_compliance_indexing(temp_result); //std::cout<<"Assessed structural performance
        (regarding compliance per space)." << std::endl ;
514
515     return temp_result ;
516 }
517
518 BSO::Building_Physics::BP_Building_Results Calculate_BP(BSO::Spatial_Design::MS_Building&
    Building )
519 {
520     BSO::Building_Physics::BP_Building_Results temp_result ;
521     static int BP_calc_number = 0 ; BP_calc_number++ ;
522
523     BSO::Spatial_Design::MS_Conformal CF( Building, &(BSO::Grammar::grammar_2) ) ;
524     //std::cout << "Made the building spatial design conformal. "<<std::endl ;
525
526     BSO::Building_Physics::BP_Simulation BP_building( CF ) ; //std::cout << "Created the
```

```
        building physics model." << std::endl ;
527
528     BP_building.sim_period() ;// std::cout << "Simulated the thermal properties of the
        building." << std::endl ;
529
530     temp_result = BP_building.get_results() ;
531
532     BSO::BP_thermal_demand_indexing(temp_result) ; //std::cout << "Assessed thermal performance.
        - " << temp_result.m_total_energy << std::endl ;
533
534     return temp_result ;
535 }
```

Heuristic Building Optimization Settings

```cpp
#ifndef HBO_SETTINGS_HPP
#define HBO_SETTINGS_HPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/Trim_And_Cast.hpp>

#ifdef SD_ANALYSIS_HPP
#include <BSO/Structural_Design/SD_Analysis.hpp>
#endif // SD_ANALYSIS_HPP

#ifdef BP_SIMULATION_HPP
#include <BSO/Building_Physics/BP_Results.hpp>
#endif // BP_SIMULATION_HPP

#include <iostream>
#include <vector>
#include <sstream>
#include <cstdlib>
#include <fstream>

namespace BSO { namespace HBO{

    enum class discipline{ ALL, SD, BP, ARG_COUNT } ; // the disciplines implemented currently
    enum class sd_compliance{ HIGH, LOW, ARG_COUNT } ; // High considers a high compliance as optimal, low considers a low compliance as optimal

    //Performance_Evaluation
    enum class assessment_level{ INDIVIDUAL, CLUSTERS, ARG_COUNT } ; // Whether spaces should be evaluated individually or in clusters
    enum class cluster_type{ KMEANS, FACADE, ARG_COUNT } ; // Determines the way clusters are constructed. KMEANS uses the k-means algorithm, FACADE constructs clusters based on the facade of a spatial design
    enum class performance_assessment{ INDIVIDUAL, AGGREGATE, NON_AGGREGATE, ARG_COUNT } ; // Evaluate by aggregating or multiple performances, CARE individual is ment for running the toolbox with either SD or BP discipline
    enum class aggregate_disciplines{ SUMMATION, PRODUCT, DISTANCE, INVERSE_PRODUCT, ARG_COUNT } ; // Which aggregation function should be used

    // Space_Selection
    enum class space_ranking{ BEST, WORST, BEST_DISTANCE, WORST_DISTANCE, ALTER_BEST, ALTER_WORST, ARG_COUNT } ;// In what way the spaces are ranked
    enum class space_removal_type{ ONE_WORST, /*TENTH, TWENTIETH, */THIRTIETH, FORTIETH, FIFTIETH, /* TOP_FLOOR, */ ARG_COUNT } ; // How many and which spaces are removed

    // Building_Modification
    enum class building_modification{ SCALE, SWEEP, ARG_COUNT } ; // Building modification techniques
    enum class rescaling_options{ X, Y, Z, XY, XZ, YZ, XYZ, ARG_COUNT } ; // Which axis scale
    enum class sweeping_options { X_Z, X_F, X_H, Y_Z, Y_F, Y_H, Z_Z, Z_F, Z_H, ARG_COUNT }; // Sweeping options. First letter is the axis, second the location
    enum class splitting options{ LARGEST, /* SMALLEST, */ BEST, WORST, ARG COUNT} ; // Which spaces to split to reach the initial space count again
    enum class splitting preference{ XYZ, XZY, YXZ, YZX, ZXY, ZYX, ARG COUNT } ; // In a space with equal sizes for 2 or 3 dimensions, which is split first

    // Old options, are replaced by sweeping options, however they can still be used if a specific sweep is requested or tested
    enum class sweeping_options_old{ X, Y, Z, ARG_COUNT } ; // Which axis to sweep;
    enum class sweeping_location{ ZERO, FIFTY, HUNDRED, ARG_COUNT }; // At what percentage of the building the sweep is executed


    // This structure contains the possible settings and data required for the optimization op a building spatial design
    struct Settings
    {
    public:

        discipline discipline_involved ;
        int cycles ;

        assessment_level individual_or_clus ;
        cluster_type clus_type ;
        performance_assessment assessment_options ;
        aggregate_disciplines aggregate_options ;

        space_ranking ranking_type ;
        space_removal_type removal_type ;

        building_modification modification_options ;
        rescaling_options rescaling ;
        sweeping_options sweep_option ;
        sweeping_location sweep_location ;
```

```
69
70          splitting_options split ;
71          splitting_preference split_pref ;
72
73          #ifdef SD_ANALYSIS_HPP
74          BSO::Structural_Design::SD_Building_Results sd_results ;
75          #endif // SD_ANALYSIS_HPP
76
77          #ifdef BP_SIMULATION_HPP
78          BSO::Building_Physics::BP_Building_Results bp_results ;
79          #endif // BP_SIMULATION_HPP
80
81          std::vector<double> weights ; // weight factors for the different performances
82          unsigned int space_removal_requested ; // the amount of spaces to be selected for
            removal, can increase due to spaces with similar performance
83          unsigned int space_removal_selected; // the amount of spaces selected for removal
84          bool inverse_SD ; // if true the spaces with low compliance are rated highly, if false
            the spaces with high compliance are rated highly
85
86          double realistic_building = 1 ; // the level of how realistic a building design is,
            number is the ratio between realstic and unrealistic spaces with 1 being realistic and 0
            unrealistic
87
88          // modifications are not always capable of continueing due to e.g. to few spaces to
            split.
89          // if so they provide the value true and the output file provides whether a space is
            terminated before the given end
90          bool terminate_simulation = false ;
91
92          Settings() // default settings
93          {
94              discipline_involved = discipline::ALL ;
95              individual_or_clus = assessment_level::INDIVIDUAL ;
96              assessment_options = performance_assessment::AGGREGATE ;
97              inverse_SD = false ;
98              aggregate_options = aggregate_disciplines::SUMMATION ;
99
100             ranking_type = space_ranking::BEST ;
101             removal_type = space_removal_type::ONE_WORST ;
102             space_removal_requested = 8 ;
103
104             modification_options = building_modification::SWEEP ;
105             split = splitting_options::LARGEST ;
106             split_pref = splitting_preference::XYZ ;
107
108         }
109         void write_file( std::string output_file ) ; // creates an outputfile which display the
            settings
110         void cout_file( ) ; // gives a cout command displaying all settings
111         void set_cycles( BSO::HBO::Performance_Evaluation::Building_Performances& build_perform
            ) ; // set an amount of cycles based on the amount of spaces in a building spatial design
112
113     }; // Settings
114
115 } // namespace HBO
116 } // namespace BSO
117 #endif // HBO SETTINGS
118
```

```
#ifndef HBO_SETTINGS_CPP
#define HBO_SETTINGS_CPP

#include <BSO/HBO/Performance_Evaluation/Building_Performances.cpp>
#include <fstream>

namespace BSO { namespace HBO{

    void Settings::write_file( std::string output_file )
    {
        std::ofstream output;
        output.open( output_file.c_str(), std::ofstream::out | std::ofstream::app );

        output << "Discipline: " << (int) discipline_involved << std::endl
               << "Assessment level: " << (int)individual_or_clus << std::endl
               << "Assessment option: " << (int)assessment_options << std::endl;

        if ( assessment_options == performance_assessment::AGGREGATE ) // if the performances
        are aggregated print the aggregate method
            output << "Aggregate option: " << (int)aggregate_options << std::endl ;
        if ( inverse_SD )
            output << "Inverse SD: 1" << std::endl ;
        output << "Space ranking: " << (int)ranking_type << std::endl
               << "Space removal type: " << (int)removal_type << std::endl
               << "Modification option: " << (int)modification_options << std::endl;
        if ( modification_options == building_modification::SCALE )
            output << "Scaling option: " << (int)rescaling << std::endl;
        else if ( modification_options == building_modification::SWEEP )
        {
            output << "Sweeping option: " << (int)sweep_option << std::endl
                   << "Sweeping location: " << (int)sweep_location << std::endl;
        }
        output << "Splitting option: " << (int)split << std::endl;
    }   // write_file()

    void Settings::cout_file( )
    {
        std::cout <<  "Discipline: " << (int ) discipline_involved << std::endl
                  <<  "Assessment level: " << (int) individual_or_clus << std::endl ;
        if ( individual_or_clus == assessment_level::CLUSTERS )
            std::cout << "Cluster type: " << ( int ) clus_type << std::endl ;
        std::cout <<  "Assessment option: " << ( int ) assessment_options << std::endl ;

        if ( assessment_options == performance_assessment::AGGREGATE )
            std::cout << "Aggregate option: " << ( int ) aggregate_options << std::endl ;
        if ( inverse_SD )
            std::cout << "Compliance: Min Best " << std::endl ;

        std::cout << "Space ranking: " << ( int ) ranking_type << std::endl
                  << "Space removal type: " << ( int ) removal_type << std::endl
                  << "Modification option: " << ( int ) modification options << std::endl ;
        if ( modification options == building modification::SWEEP )
            std::cout << "Sweep option: " << ( int ) sweep option << std::endl ;
        if ( modification options == building modification::SCALE )
            std::cout << "Scaling option: " << ( int ) rescaling << std::endl ;
        std::cout << "Splitting option: " << ( int ) split << std::endl
                  << "Splitting preference: " << ( int ) split pref << std::endl ;
    }

    void Settings::set_cycles( BSO::HBO::Performance_Evaluation::Building_Performances&
    build_perform )
    {
        switch( removal_type )
        {
            case space_removal_type::ONE_WORST:
            {
                if ( individual_or_clus == assessment_level::CLUSTERS )
                {
                    cycles = ceil( (float) build_perform.get_initial_space_count() / 10 ) ;
                }
                else
                {
                cycles = build_perform.groups.size() ;
                }
                break ;
            }
            //case space_removal_type::TENTH: { cycles =
            (float)build_perform.get_initial_space_count() / ceil(
            build_perform.get_initial_space_count() * 0.10 ) ; break ; }
            //case space_removal_type::TWENTIETH: { cycles =
            (float)build_perform.get_initial_space_count() / ceil(
            build_perform.get_initial_space_count() * 0.20 ) ; break ; }
            case space_removal_type::THIRTIETH: { cycles =
            (float)build_perform.get_initial_space_count() / ceil(
```

```
                build_perform.get_initial_space_count() * 0.30 ) ; break ; }
78              case space_removal_type::FORTIETH: { cycles =
                (float)build_perform.get_initial_space_count() / ceil(
                build_perform.get_initial_space_count() * 0.40 ) ; break ; }
79              case space_removal_type::FIFTIETH: { cycles =
                (float)build_perform.get_initial_space_count() / ceil(
                build_perform.get_initial_space_count() * 0.50 ) ; break ; }
80              //case space_removal_type::TOP_FLOOR: { cycles = 2  ; break ;   }
81              case space_removal_type::ARG_COUNT: { std::cout << "Error, ARG_COUNT reached for
                removal ( HBO_Settings.hpp ) " << std::endl ; exit(1) ; }
82              default: { std::cout << "Error default, exiting now... (HBO_Settings.hpp)" <<
                std::endl ; exit(1) ; }
83          }
84      }
85  } // namespace HBO
86  } // namespace BSO
87  #endif // HBO_SETTINGS
88
```

Building Performances Class

```
#ifndef BUILDING_PERFORMANCES_HPP
#define BUILDING_PERFORMANCES_HPP

namespace BSO { namespace HBO { namespace Performance_Evaluation
{

    class Building_Performances;


} // namespace Performance_Evaluation
} // namespace HBO
} // namespace BSO

#endif // BUILDING_PERFORMANCES

```

```cpp
#ifndef BUILDING_PERFORMANCES_CPP
#define BUILDING_PERFORMANCES_CPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>

#ifdef SD_ANALYSIS_HPP
#include <BSO/Structural_Design/SD_Analysis.hpp>
#endif // SD_ANALYSIS_HPP

#ifdef BP_SIMULATION_HPP
#include <BSO/Building_Physics/BP_Results.hpp>
#endif // BP_SIMULATION_HPP

#include <BSO/Data.hpp>
#include <BSO/Clustering.hpp>
#include <BSO/Trim_And_Cast.hpp>

#include <BSO/HBO/HBO_Settings.hpp>

#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>

namespace BSO { namespace HBO { namespace Performance_Evaluation
{
    /* Data structures to process performance evaluations */

    struct group_performance
    {
        std::vector<int> space_ID; // all space IDs that belong to this group
        std::vector<double> initial_performance; // stores the initial performances as calculated for the space(s) belonging to this group
        std::vector<double> modified_performance; // the modified performances belonging to this group, modified can be e.g. aggregated or with additional weights

        bool id_in_space(int& id) // checks if a space ID belongs to this group object
        {
            for ( unsigned int i = 0 ; i < space_ID.size() ; i++ )
            {
                if (space_ID[i] == id ) return true;
            }

            return false;
        }

        double best_modified_performance() // returns the best modified performance that this groups has
        {
            double best_perform = modified_performance[0];
            for (unsigned int i = 0 ; i < modified performance.size() ; i++ )
            {
                if ( modified performance[i] > best perform )
                {
                    best perform = modified performance[i];
                }
            }
            return best perform;
        }

        double worst_modified_performance() // returns the worst performance that this group has
        {
            double worst_perform = modified_performance[0];
            for ( unsigned int i = 0 ; i < modified_performance.size() ; i++ )
            {
                if( modified_performance[i] < worst_perform )
                {
                    worst_perform = modified_performance[i];
                }
            }
            return worst_perform;
        }
        ~group_performance() {}
    };

    class Building_Performances
    {
    private:
        HBO::assessment_level assess_lvl ;
        double initial_volume ; // stores the initial volume of the spatial design
        int initial_space_count ; // stores the initial space count of the spatial design

    public:
        Building_Performances() ; // ctor, creates an empty object
```

```
82          Building_Performances( BSO::Spatial_Design::MS_Building&, Settings& ) ;
83
84          Building_Performances( BSO::Spatial_Design::MS_Building&, Building_Performances& ) ; //
            ctor that is used when spaces are deleted so the 'old' Building_Performances is not up
            to date
85          ~Building_Performances(); // dtor
86
87          std::vector<group_performance> groups ; // contains performance information of the spaces
88
89          void Create_Facade_Clusters( BSO::Spatial_Design::MS_Building& ) ; // creates clusters
            of the spaces at the facade of the building, NOTE spaces can belong to multiple clusters
90          void add_group( group_performance ) ; // adds a group to the group vector
91          void delete_group( int group_index ) ; // remove the group at the group index
92
93          //std::vector<double> weights; // contains weights for the different disciplines, can be
            empty. Should not be used, check HBO::HBO_Settings for weights
94
95          int id_count() ; // returns the total amount of space_IDs over all groups
96          int id_belonging_group( int& ) ; // searches the space vector for the id asked and
            returns the index of the object in the vector
97          int best_group_initial() ; //  returns the index with the best initial performance
98          int best_group_modified() ; //  returns the index with the best modified performance
99          int worst_group_initial() ; // returns the space index with worst initial performance
100         int worst_group_modified() ; // return the space index with worst modified performance
101
102         int best_group_distance() ; // return the space index for the group with the shortest
            distance to (1,1)
103         int worst_group_distance() ; // return the space index for the group with the shortest
            distance to (0,0)
104
105         double get_initial_volume() ; // returns the initial volume
106         int get_initial_space_count() ; // returns the initial space count
107
108         //void print_build_perform(); // prints the content of this class to a .txt file
109     };
110
111     Building_Performances::Building_Performances()
112     {
113
114     } //ctor
115
116     Building_Performances::Building_Performances( BSO::Spatial_Design::MS_Building&
        current_design, Settings& settings )
117     {
118         #ifdef SD_ANALYSIS_HPP
119         #ifdef BP_SIMULATION_HPP
120         if ( settings.discipline_involved == HBO::discipline::ALL )
121         {
122             initial_space_count = current_design.obtain_space_count();
123             initial_volume = current_design.get_volume();
124
125             switch(settings.individual or clus)
126             {
127                 case assessment level::INDIVIDUAL:
128                     {
129                         // for all spaces in ms building
130                     for( int i = 0 ; i < current design.obtain space count() ; i++ )
131                     {
132                         group_performance temp_space;
133                         temp_space.space_ID.push_back(current_design.obtain_space(i).ID);
134
135                         // search the sd result belonging to it
136                         for(unsigned int j = 0 ; j < settings.sd_results.m_spaces.size() ; j++ )
137                         {
138                             if(temp_space.space_ID.back() == settings.sd_results.m_spaces[j].m_ID)
139                             {
140                                 if ( settings.inverse_SD == false )
141                                 {
142
                                        temp_space.initial_performance.push_back(settings.sd_results.m
                                        _spaces[j].m_rel_performance);
143                                 }
144                                 else if ( settings.inverse_SD == true )
145                                 {
146                                     temp_space.initial_performance.push_back( 1 -
                                        settings.sd_results.m_spaces[j].m_rel_performance) ;
147                                 }
148                                 else {std::cout<<"Error, in assinging SD performance.
                                    Building_Performances.hpp, exiting now... "<<std::endl; exit(1);}
149
150                                 break;
151                             }
152                         } // end sd_result
153
```

```
154                         // and the bp result belonging to it
155                         for(unsigned int j = 0 ; j < settings.bp_results.m_space_results.size()
                            ; j++)
156                         {
157
158                             if ( temp_space.space_ID.back() ==
                                BSO::trim_and_cast_int(settings.bp_results.m_space_results[j].m_space_
                                ID))
159                             {
160
                                    temp_space.initial_performance.push_back(settings.bp_results.m_spa
                                    ce_results[j].m_rel_performance);
161                                 break;
162                             }
163                         } // end bp_result
164
165                         groups.push_back(temp_space);
166                     }
167
168                     // check whether all groups have the same amount of performance values
169                     for ( unsigned int i = 0 ; i < groups.size() -1 ; i++ )
170                     {
171                         if (groups[i].initial_performance.size() ==
                            groups[i+1].initial_performance.size() )
172                         {
173                             continue;
174                         }
175                         else if( groups[i].initial_performance.size() !=
                            groups[i+1].initial_performance.size() )
176                         {
177                             std::cout<< "Error, amount of performance values not equal...
                                exiting now ( Building_Performances.hpp)";
178                             exit(1);
179                         }
180                         else
181                         {
182                             std::cout<<"Error in amount of spaces... exiting now
                                (Building_Performances.hpp)";
183                             exit(1);
184                         }
185                     }
186
187                     break;
188                     }
189
190                 case assessment_level::CLUSTERS:
191                     {
192                         switch( settings.clus_type )
193                         {
194                             case HBO::cluster_type::KMEANS:
195                             {
196                                 assess lvl = assessment level::CLUSTERS;
197                                 // get the results and put them in a vector (shared ptr for
                                    clustering purposes)
198                                 std::shared ptr<std::vector<BSO::data point> > data set =
                                    std::make shared<std::vector<BSO::data point> >();
199                                 std::vector<int> space id list; // to track space ID's across
                                    result structures of different disciplines
200                                 unsigned int number_of_spaces =
                                    current_design.obtain_space_count(); // determine how many
                                    spaces are in the design
201
202                                 for (unsigned int i = 0; i < number_of_spaces; i++)
203                                 { // for each space in the design
204                                     // initialise a data point and an ID
205                                     space_id_list.push_back(current_design.obtain_space(i).ID);
206                                     BSO::data_point temp = Eigen::Vector2d::Zero();
207                                     data_set->push_back(temp);
208                                 }
209
210                                 bool match_check[space_id_list.size()] = {false}; // this list
                                    will keep track if a performance has been found for each space
211
212                                 for (unsigned int i = 0; i <
                                    settings.bp_results.m_space_results.size(); i++)
213                                 { // for each space result
214                                     for (unsigned int j = 0; j < number_of_spaces; j++)
215                                     { // and for each space in the design
216                                         if (space_id_list[j] ==
                                            BSO::trim_and_cast_int(settings.bp_results.m_space_results
                                            [i].m_space_ID))
217                                         { // check if space_result matches with a space in the
                                            design
218                                             (*data_set)[j](0) =
```

```
                                settings.bp_results.m_space_results[i].m_total_energy;
219                             match_check[j] = true;
220                         }
221                     }
222                 }
223
224                 for (unsigned int i = 0; i < number_of_spaces; i++)
225                 { // for each space in the design
226                     if (!match_check[i])
227                     { // check if a performance has been found for space i
228                         std::cout << "Error could not find BP performance of
                            space with ID: "
229                                   << space_id_list[i] << "
                                      (Building_Performances.hpp), exiting now..."
                                      << std::endl;
230                         exit(1);
231                     }
232                 }
233
234                 for (unsigned int i = 0; i <
                    settings.sd_results.m_spaces.size(); i++)
235                 { // for each space result
236                     for (unsigned int j = 0; j < space_id_list.size(); j++)
237                     { // and for each space in the design
238                         if (space_id_list[j] ==
                            settings.sd_results.m_spaces[i].m_ID)
239                         { // check if space_result matches with a space in the
                            design
240                             (*data_set)[j](1) =
                                settings.sd_results.m_spaces[i].m_total_compliance;
241                             match_check[j] = false;
242                         } //since the BP check set the values to true SD needs
                            to switch them back to false
243                     }
244                 }
245
246                 for (unsigned int i = 0; i < number_of_spaces; i++)
247                 { // for each space in the design
248                     if (match_check[i])
249                     { // check if a performance has been found for space i
250                         std::cout << "Error could not find structural
                            performance of space with ID: "
251                                   << space_id_list[i] << "
                                      (Building_Performances.hpp), exiting now..."
                                      << std::endl;
252                         exit(1);
253                     }
254                 }
255
256                 // cluster the data set
257                 // make k clusters, where k lies between 25% and 75% of the
                    number of spaces with a lower limit of 2 clusters
258                 //std::vector<std::shared ptr<BSO::Cluster> > clustered data set
                    = clustering(data set, 50, (2 < 1+number of spaces/8)?
                    1+number of spaces/4 : 2, 3*number of spaces/4, 2);
259                 std::vector<std::shared ptr<BSO::Cluster> > clustered data set =
                    BSO::clustering(data set, 50, 6, 10, 2);
260
261                 for ( unsigned int i = 0 ; i < clustered_data_set.size() ; i++ )
262                 {
263                     for ( unsigned int j = 0 ; j <
                        current_design.obtain_space_count() ; j++ )
264                     {
265                         if ( clustered_data_set[i]->m_bit_mask[j] == 1)
266                         {
267
                                clustered_data_set[i]->m_space_id_list.push_back(curre
                                nt_design.obtain_space(j).ID);
268                         }
269                     }
270                 }
271
272                 for (unsigned int i = 0 ; i < clustered_data_set.size() ; i++ )
273                 { // for all clusters
274
275                     group_performance temp_group;
276
277                     for(unsigned int j = 0 ; j <
                        clustered_data_set[i]->m_space_id_list.size() ; j++ )
278                     { // add space ids to the space performance
279
                            temp_group.space_ID.push_back(clustered_data_set[i]->m_spa
                            ce_id_list[j]);
280                     }
```

```
281                                 for(unsigned int j = 0 ; j <
                                    clustered_data_set[i]->m_centroid.size() ; j++ )
282                                 { // add centroid to initial performance
283
                                        temp_group.initial_performance.push_back(clustered_data_se
                                        t[i]->m_centroid[j]);
284                                 }
285
286                                 groups.push_back(temp_group);
287                             }
288
289                             break;
290                         }
291                         case HBO::cluster_type::FACADE:
292                         {
293                             this->Create_Facade_Clusters( current_design ) ;
294
295                             // find all space performances belonging to the groups
296
297                             for ( unsigned int i = 0 ; i < groups.size() ; i++ )
298                             {
299                                 for ( unsigned int j = 0 ; j < groups[ i ].space_ID.size() ;
                                    j++ )
300                                 {
301                                     for ( unsigned int k = 0 ; k <
                                        settings.sd_results.m_spaces.size() ; k++ )
302                                     {
303                                         if ( settings.sd_results.m_spaces[ k ].m_ID ==
                                            groups[ i ].space_ID[ j ] )
304                                         {
305                                             groups[ i ].initial_performance.push_back(
                                                settings.sd_results.m_spaces[ k
                                                ].m_rel_performance ) ;
306                                             break ;
307                                         }
308                                     }
309                                     for ( unsigned int k = 0 ; k <
                                        settings.bp_results.m_space_results.size() ; k++ )
310                                     {
311                                         if ( BSO::trim_and_cast_int(
                                            settings.bp_results.m_space_results[ k ].m_space_ID
                                            ) == groups[ i ].space_ID[ j ] )
312                                         {
313                                             groups[ i ].initial_performance.push_back(
                                                settings.bp_results.m_space_results[ k
                                                ].m_rel_performance ) ;
314                                             break ;
315                                         }
316                                     }
317                                 }
318                             }
319
320                             // average all performances of the groups
321                             for ( unsigned int i = 0 ; i < groups.size() ; i++ )
322                             {
323                                 double average value = 0 ;
324                                 for ( unsigned int j = 0 ; j < groups[ i
                                    ].initial performance.size() ; j++ )
325                                 {
326                                     average_value += groups[ i ].initial_performance[ j ] ;
327                                 }
328                                 average_value = average_value / groups[ i ].space_ID.size() ;
329                                 groups[ i ].initial_performance.clear() ;
330                                 groups[ i ].initial_performance.push_back( average_value );
331                             }
332
333                             break ;
334                         }
335                         case HBO::cluster_type::ARG_COUNT:
336                         {
337                             std::cout<< "Error ARG_COUNT in cluster_type, exiting now... (
                                Building_Performances.cpp )" << std::endl ;
338                             exit( 1 ) ;
339                             break ;
340                         }
341                         default:
342                         {
343                             std::cout << "Error default in cluster_type, exiting now... (
                                Building_Performances.cpp) " << std::endl ;
344                             exit( 1 ) ;
345                         }
346                     }
347                     break ;
348                 }
```

```
349                 case assessment_level::ARG_COUNT:
350                     {
351                         std::cout << "Error in Settings::assessment_level, exiting now.... (
                            Building_Performances.cpp )" << std::endl ;
352                         exit( 1 ) ;
353                         break ;
354                     }
355
356                 default:
357                     {
358                         std::cout << "Error in ctor Building_Performances, exiting now.... (
                            Building_Performances.cpp )" << std::endl ;
359                         exit( 1 ) ;
360                     }
361             } // switch(individual_or_cluster
362         }
363         #endif // BP_SIMULATION_HPP
364         #endif // SD_ANALYSIS_HPP
365
366         #ifdef SD_ANALYSIS_HPP
367         if ( settings.discipline_involved == HBO::discipline::SD )
368         {
369             initial_space_count = current_design.obtain_space_count();
370             initial_volume = current_design.get_volume();
371
372             switch(settings.individual_or_clus)
373             {
374                 case assessment_level::INDIVIDUAL:
375                     {
376                         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
377                         {
378                             group_performance temp_group;
379                             temp_group.space_ID.push_back(current_design.obtain_space(i).ID);
380
381                             for ( unsigned int j = 0 ; j < settings.sd_results.m_spaces.size() ;
                                j++ )
382                             {
383                                 if ( settings.sd_results.m_spaces[j].m_ID ==
                                    temp_group.space_ID.back() )
384                                 {
385                                     if ( settings.inverse_SD == false )
386                                         temp_group.initial_performance.push_back(
                                            settings.sd_results.m_spaces[ j ].m_rel_performance);
387                                     else if ( settings.inverse_SD == true )
388                                         temp_group.initial_performance.push_back( 1 -
                                            settings.sd_results.m_spaces[ j ].m_rel_performance ) ;
389                                     else { std::cout<< "Error in building_performances.hpp, sd
                                        contructor. Exiting now.. " << std::endl; exit(1) ; }
390                                     break;
391                                 }
392                             }
393                             groups.push back(temp group);
394                         }
395                         break;
396                     }
397                 case assessment level::CLUSTERS:
398                     {
399                         switch( settings.clus type )
400                         {
401                             case HBO::cluster_type::KMEANS:
402                             {
403                                 assess_lvl = assessment_level::CLUSTERS;
404                                 // get the results and put them in a vector (shared ptr for
                                    clustering purposes)
405                                 std::shared_ptr<std::vector<BSO::data_point> > data_set =
                                    std::make_shared<std::vector<BSO::data_point> >();
406                                 std::vector<int> space_id_list; // to track space ID's across
                                    result structures of different disciplines
407                                 unsigned int number_of_spaces =
                                    current_design.obtain_space_count(); // determine how many
                                    spaces are in the design
408
409                                 for (unsigned int i = 0; i < number_of_spaces; i++)
410                                 { // for each space in the design
411                                     // initialise a data point and an ID
412                                     space_id_list.push_back(current_design.obtain_space(i).ID);
413                                     BSO::data_point temp = Eigen::Vector2d::Zero();
414                                     data_set->push_back(temp);
415                                 }
416
417                                 bool match_check[space_id_list.size()] = {false}; // this list
                                    will keep track if a performance has been found for each space
418
419                                 for (unsigned int i = 0; i <
```

```
                    settings.sd_results.m_spaces.size(); i++)
                    { // for each space result
                        for (unsigned int j = 0; j < space_id_list.size(); j++)
                        { // and for each space in the design
                            if (space_id_list[j] ==
                            settings.sd_results.m_spaces[i].m_ID)
                            { // check if space_result matches with a space in the
                            design
                                (*data_set)[j](0) =
                                settings.sd_results.m_spaces[i].m_total_compliance ;
                                (*data_set)[j](1) = 1 ;
                                match_check[j] = true;
                            } //since the BP check set the values to true SD needs
                            to switch them back to false
                        }
                    }

                    for (unsigned int i = 0; i < number_of_spaces; i++)
                    { // for each space in the design
                        if (!match_check[i])
                        {   // check if a performance has been found for space i
                            std::cout << "Error could not find structural
                            performance of space with ID: "
                                      << space_id_list[i] << "
                                      (Building_Performances.hpp), exiting now..."
                                      << std::endl;
                            exit(1);
                        }
                    }

                    // cluster the data set
                    // make k clusters, where k lies between 25% and 75% of the
                    number of spaces with a lower limit of 2 clusters
                    //std::vector<std::shared_ptr<BSO::Cluster> > clustered_data_set
                    = clustering(data_set, 50, (2 < 1+number_of_spaces/8)?
                    1+number_of_spaces/4 : 2, 3*number_of_spaces/4, 2);
                    std::vector<std::shared_ptr<BSO::Cluster> > clustered_data_set =
                    BSO::clustering(data_set, 50, 6, 10, 1);

                    for ( unsigned int i = 0 ; i < clustered_data_set.size() ; i++ )
                    {
                        for ( unsigned int j = 0 ; j <
                        current_design.obtain_space_count() ; j++ )
                        {
                            if ( clustered_data_set[i]->m_bit_mask[j] == 1)
                            {

                                clustered_data_set[i]->m_space_id_list.push_back(curre
                                nt_design.obtain_space(j).ID);
                            }
                        }
                    }

                    for (unsigned int i = 0 ; i < clustered data set.size() ; i++ )
                    { // for all clusters

                        group performance temp group;

                        for(unsigned int j = 0 ; j <
                        clustered_data_set[i]->m_space_id_list.size() ; j++ )
                        { // add space ids to the space performance

                            temp_group.space_ID.push_back(clustered_data_set[i]->m_spa
                            ce_id_list[j]);
                        }
                        for(unsigned int j = 0 ; j <
                        clustered_data_set[i]->m_centroid.size() ; j++ )
                        { // add centroid to initial performance

                            temp_group.initial_performance.push_back(clustered_data_se
                            t[i]->m_centroid[j]);
                        }

                        groups.push_back(temp_group);
                    }

                    break;
                }
                case HBO::cluster_type::FACADE:
                {
                    this->Create_Facade_Clusters( current_design ) ;

                    // find all space performances belonging to the groups

```

```
                                for ( unsigned int i = 0 ; i < groups.size() ; i++ )
                                {
                                    for ( unsigned int j = 0 ; j < groups[ i ].space_ID.size() ;
                                    j++ )
                                    {
                                        for ( unsigned int k = 0 ; k <
                                        settings.sd_results.m_spaces.size() ; k++ )
                                        {
                                            if ( settings.sd_results.m_spaces[ k ].m_ID ==
                                            groups[ i ].space_ID[ j ] )
                                            {
                                                groups[ i ].initial_performance.push_back(
                                                settings.sd_results.m_spaces[ k
                                                ].m_rel_performance ) ;
                                                break ;
                                            }
                                        }
                                    }
                                }

                                // average all performances of the groups
                                for ( unsigned int i = 0 ; i < groups.size() ; i++ )
                                {
                                    double average_value = 0 ;
                                    for ( unsigned int j = 0 ; j < groups[ i
                                    ].initial_performance.size() ; j++ )
                                    {
                                        average_value += groups[ i ].initial_performance[ j ] ;
                                    }
                                    average_value = average_value / groups[ i ].space_ID.size() ;
                                    groups[ i ].initial_performance.clear() ;
                                    groups[ i ].initial_performance.push_back( average_value );
                                }

                                break ;
                            }
                            case HBO::cluster_type::ARG_COUNT:
                            {
                                std::cout<< "Error cluster_type::ARG_COUNT reached, exiting
                                now... ( Building_Performances.cpp)" << std::endl ;
                                exit(1) ;
                            }
                        }
                        break ;
                    }
                case assessment_level::ARG_COUNT:
                    {
                        std::cout << "Error in Settings::assessment_level, exiting now....
                        (Building_Performances.hpp)" << std::endl ;
                        exit(1) ;
                        break ;
                    }
                default:
                    {
                        std::cout << "Error in ctor Building Performances, exiting now.... (
                        Building Performances.hpp)" << std::endl ;
                        exit(1) ;
                        break ;
                    }
            } //switch(individual_or_cluster)
        }
        #endif // SD_ANALYSIS_HPP

        #ifdef BP_SIMULATION_HPP
        if ( settings.discipline_involved == HBO::discipline::BP )
        {
            initial_space_count = current_design.obtain_space_count();
            initial_volume = current_design.get_volume();

            switch(settings.individual_or_clus)
            {
                case assessment_level::INDIVIDUAL:
                    {
                        for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
                        {
                            group_performance temp_group;
                            temp_group.space_ID.push_back(current_design.obtain_space(i).ID);

                            for ( unsigned int j = 0 ; j <
                            settings.bp_results.m_space_results.size() ; j++ )
                            {
                                if ( BSO::trim_and_cast_int(settings.bp_results.m_space_results[
                                j ].m_space_ID) == temp_group.space_ID[ 0 ] )
                                {
```

```
556                                 temp_group.initial_performance.push_back(
                                    settings.bp_results.m_space_results[j].m_rel_performance );
557                                 break ;
558                             }
559                         }
560                         groups.push_back(temp_group);
561                     }
562                     break;
563                 }
564             case assessment_level::CLUSTERS:
565                 {
566                     switch ( settings.clus_type )
567                     {
568                         case HBO::cluster_type::KMEANS:
569                         {
570                             assess_lvl = assessment_level::CLUSTERS;
571                             // get the results and put them in a vector (shared ptr for
                                clustering purposes)
572                             std::shared_ptr<std::vector<BSO::data_point> > data_set =
                                std::make_shared<std::vector<BSO::data_point> >();
573                             std::vector<int> space_id_list; // to track space ID's across
                                result structures of different disciplines
574                             unsigned int number_of_spaces =
                                current_design.obtain_space_count(); // determine how many
                                spaces are in the design
575
576                             for (unsigned int i = 0; i < number_of_spaces; i++)
577                             { // for each space in the design
578                                 // initialise a data point and an ID
579                                 space_id_list.push_back(current_design.obtain_space(i).ID);
580                                 BSO::data_point temp = Eigen::Vector2d::Zero();
581                                 data_set->push_back(temp);
582                             }
583
584                             bool match_check[space_id_list.size()] = {false}; // this list
                                will keep track if a performance has been found for each space
585
586                             for (unsigned int i = 0; i <
                                settings.bp_results.m_space_results.size(); i++)
587                             { // for each space result
588                                 for (unsigned int j = 0; j < number_of_spaces; j++)
589                                 { // and for each space in the design
590                                     if (space_id_list[j] ==
                                        BSO::trim_and_cast_int(settings.bp_results.m_space_results
                                        [i].m_space_ID))
591                                     { // check if space_result matches with a space in the
                                        design
592                                         (*data_set)[j](0) =
                                            settings.bp_results.m_space_results[i].m_rel_performan
                                            ce;
593                                         match check[j] = true;
594                                     }
595                                 }
596                             }
597
598                             for (unsigned int i = 0; i < number of spaces; i++)
599                             { // for each space in the design
600                                 if (!match check[i])
601                                 { // check if a performance has been found for space i
602                                     std::cout << "Error could not find BP performance of
                                        space with ID: "
603                                               << space_id_list[i] << "
                                                (Building_Performances.hpp), exiting now..."
                                               << std::endl;
604                                     exit(1);
605                                 }
606                             }
607
608                             // cluster the data set
609                             // make k clusters, where k lies between 25% and 75% of the
                                number of spaces with a lower limit of 2 clusters
610                             //std::vector<std::shared_ptr<BSO::Cluster> > clustered_data_set
                                = clustering(data_set, 50, (2 < 1+number_of_spaces/8)?
                                1+number_of_spaces/4 : 2, 3*number_of_spaces/4, 2);
611                             std::vector<std::shared_ptr<BSO::Cluster> > clustered_data_set =
                                BSO::clustering(data_set, 50, 6, 10, 1);
612
613                             for ( unsigned int i = 0 ; i < clustered_data_set.size() ; i++ )
614                             {
615                                 for ( unsigned int j = 0 ; j <
                                    current_design.obtain_space_count() ; j++ )
616                                 {
617                                     if ( clustered_data_set[i]->m_bit_mask[j] == 1)
618                                     {
```

```
                                clustered_data_set[i]->m_space_id_list.push_back(curre
                                nt_design.obtain_space(j).ID);
                        }
                    }
                }

                for (unsigned int i = 0 ; i < clustered_data_set.size() ; i++ )
                { // for all clusters

                    group_performance temp_group;

                    for(unsigned int j = 0 ; j <
                    clustered_data_set[i]->m_space_id_list.size() ; j++ )
                    { // add space ids to the space performance

                        temp_group.space_ID.push_back(clustered_data_set[i]->m_spa
                        ce_id_list[j]);
                    }
                    for(unsigned int j = 0 ; j <
                    clustered_data_set[i]->m_centroid.size() ; j++ )
                    { // add centroid to initial performance

                        temp_group.initial_performance.push_back(clustered_data_se
                        t[i]->m_centroid[j]);
                    }

                    groups.push_back(temp_group);
                }

                break;
            }
            case HBO::cluster_type::FACADE:
            {
                this->Create_Facade_Clusters( current_design ) ;

                // find all space performances belonging to the groups

                for ( unsigned int i = 0 ; i < groups.size() ; i++ )
                {
                    for ( unsigned int j = 0 ; j < groups[ i ].space_ID.size() ;
                    j++ )
                    {
                        for ( unsigned int k = 0 ; k <
                        settings.bp_results.m_space_results.size() ; k++ )
                        {
                            if ( BSO::trim_and_cast_int(
                            settings.bp_results.m_space_results[ k ].m_space_ID
                            ) == groups[ i ].space_ID[ j ] )
                            {
                                groups[ i ].initial performance.push back(
                                settings.bp results.m space results[ k
                                ].m rel performance ) ;
                                break ;
                            }
                        }
                    }
                }

                // average all performances of the groups
                for ( unsigned int i = 0 ; i < groups.size() ; i++ )
                {
                    double average_value = 0 ;
                    for ( unsigned int j = 0 ; j < groups[ i
                    ].initial_performance.size() ; j++ )
                    {
                        average_value += groups[ i ].initial_performance[ j ] ;
                    }
                    average_value = average_value / groups[ i ].space_ID.size() ;
                    groups[ i ].initial_performance.clear() ;
                    groups[ i ].initial_performance.push_back( average_value );
                }

                break ;
            }
            case HBO::cluster_type::ARG_COUNT:
            {
                std::cout<< "Error ARG_COUNT reached, exiting now... (
                Building_Performances.cpp)" << std::endl ;
                exit( 1 ) ;
                break ;
            }
            default:
            {
```

```
687                                 std::cout<< "Default error, exiting now... (
                                    Building_Performances.cpp ) " << std::endl ;
688                                 exit( 1 ) ;
689                                 break ;
690                             }
691                         }
692                         break ;
693                     }
694                 case assessment_level::ARG_COUNT:
695                     {
696                         std::cout << "Error in Settings::assessment_level, exiting now....
                            (Building_Performances.hpp)" << std::endl ;
697                         exit( 1 ) ;
698                         break ;
699                     }
700                 default:
701                     {
702                             std::cout << "Error in ctor Building_Performances, exiting now.... (
                                Building_Performances.hpp)" << std::endl ;
703                             exit( 1 ) ;
704                             break ;
705                     }
706             }
707         }
708         #endif // BP_SIMULATION_HPP
709
710     }
711
712     Building_Performances::Building_Performances(BSO::Spatial_Design::MS_Building& design,
        Building_Performances& initial_perform)
713     {
714         std::vector< int > ids_present;
715
716         for ( unsigned int i = 0 ; i < design.obtain_space_count() ; i++ ) {
            ids_present.push_back(design.get_space_ID(i)); }
717
718         for ( auto it1 = initial_perform.groups.begin() ; it1 != initial_perform.groups.end() ;
            ) // iterate over all original groups
719         {
720             group_performance temp_group = *it1; // get the group performance belonging to the
                space id
721
722             for ( auto it2 = temp_group.space_ID.begin() ; it2 != temp_group.space_ID.end() ; )
                // iterate over all space ids in that group
723             {
724                bool id_removed = false;
725
726                 for ( auto it3 = ids_present.begin() ; it3 != ids_present.end() ; ) // iterate
                    over all ids in the new building
727                 {
728
729                     if ( *it2 == *it3  ) // if the space id is present in the new building,
                        search for the next id in group
730                     {
731                         ids present.erase( it3 );
732                         id removed = true;
733
734                     }
735                     else
736                     {
737                         ++it3;
738                     }
739
740                 } // end ids_present loop
741
742                 if( !id_removed )
743                 {
744                    temp_group.space_ID.erase( it2 );
745                 }
746                 else { ++it2; }
747
748             } // end temp_group loop
749             if ( !temp_group.space_ID.empty() )
750                groups.push_back(temp_group);
751             ++it1;
752         } // end groups loop
753
754     } //ctor (MS_Building, initial_build_perform)
755
756
757
758     Building_Performances::~Building_Performances()
759     {
760
```

```
761         } //dtor
762
763         void Building_Performances::Create_Facade_Clusters( BSO::Spatial_Design::MS_Building&
            current_design )
764         {
765             std::vector< group_performance > temp_groups( 7 ) ; // create 7 groups, 1 for each outer
                facade and 1 for all non facade spaces
766
767             // create cluster for  z = min
768             double min_z = current_design.obtain_space( 0 ).z ;
769             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
770             {
771                 if ( current_design.obtain_space( i ).z < min_z )
772                 {
773                     min_z = current_design.obtain_space( i ).z ;
774                 }
775             }
776
777             double height = 0 ;
778
779             // find the height of the facade elements to be removed
780             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
781             {
782                 if ( current_design.obtain_space( i ).z == min_z && current_design.obtain_space( i
                    ).height > height )
783                 {
784                     height = current_design.obtain_space( i ).height ;
785                 }
786             }
787
788             // find all spaces that are within this column
789             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
790             {
791                 if( current_design.obtain_space( i ).z == min_z ||  ( abs(
                    current_design.obtain_space( i ).z ) + current_design.obtain_space( i ).height ) <=
                    height )
792                 {
793                     temp_groups[ 0 ].space_ID.push_back( current_design.obtain_space( i ).ID ) ;
794                 }
795             }
796
797             // create cluster for z = max
798             double max_height = 0 ;
799
800             // find the top level of the building
801             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
802             {
803                 if ( current_design.obtain_space( i ).z + current_design.obtain_space( i ).height >
                    max_height )
804                 {
805                     max height = current design.obtain space( i ).z + current design.obtain space( i
                        ).height ;
806                 }
807             }
808
809             double low z = max height ;
810             for ( unsigned int i = 0 ; i < current design.obtain space count() ; i++ )
811             {
812                 if ( current_design.obtain_space( i ).z + current_design.obtain_space( i ).height ==
                    max_height && current_design.obtain_space( i ).z < low_z )
813                 {
814                     low_z = current_design.obtain_space( i ).z ;
815                 }
816             }
817
818             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
819             {
820                 if ( current_design.obtain_space( i ).z >= low_z )
821                 {
822                     temp_groups[ 1 ].space_ID.push_back( current_design.obtain_space( i ).ID ) ;
823                 }
824             }
825
826             // create cluster for x = min
827             double min_x = current_design.obtain_space( 0 ).x ;
828             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
829             {
830                 if ( current_design.obtain_space( i ).x < min_x )
831                 {
832                     min_x = current_design.obtain_space( i ).x ;
833                 }
834             }
835
836             double width = 0 ;
```

```
837         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
838         {
839             if ( current_design.obtain_space( i ).x == min_x && current_design.obtain_space( i
                ).width > width )
840             {
841                 width = current_design.obtain_space( i ).width ;
842             }
843         }
844
845         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
846         {
847             if ( current_design.obtain_space( i ).x == min_x ||  ( abs(
                current_design.obtain_space( i ).x ) + current_design.obtain_space( i ).width ) <=
                width )
848             {
849                 temp_groups[ 2 ].space_ID.push_back( current_design.obtain_space( i ).ID ) ;
850             }
851         }
852
853         // create cluster for x = max
854         double max_width = 0 ;
855         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
856         {
857             if ( current_design.obtain_space( i ).x + current_design.obtain_space( i ).width >
                max_width )
858             {
859                 max_width = current_design.obtain_space( i ).x + current_design.obtain_space( i
                    ).width ;
860             }
861         }
862
863         double max_x = max_width ;
864         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
865         {
866             if ( current_design.obtain_space( i ).x < max_x && current_design.obtain_space( i
                ).x + current_design.obtain_space( i ).width == max_width )
867             {
868                 max_x = current_design.obtain_space( i ).x ;
869             }
870         }
871
872         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
873         {
874             if ( current_design.obtain_space( i ).x >= max_x )
875             {
876                 temp_groups[ 3 ].space_ID.push_back( current_design.obtain_space( i ).ID ) ;
877             }
878         }
879
880         // create cluster for y = min
881         double min y = current design.obtain space( 0 ).y ;
882         for ( unsigned int i = 0 ; i < current design.obtain space count() ; i++ )
883         {
884             if ( current design.obtain space( i ).y < min y )
885                 min y = current design.obtain space( i ).y ;
886         }
887
888         double depth = 0 ;
889         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
890         {
891             if ( current_design.obtain_space( i ).depth > depth && current_design.obtain_space(
                i ).y == min_y )
892                 depth = current_design.obtain_space( i ).depth ;
893         }
894
895         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
896         {
897             if ( current_design.obtain_space( i ).y == min_y || ( abs(
                current_design.obtain_space( i ).y ) + current_design.obtain_space( i ).depth <=
                depth ) )
898                 temp_groups[ 4 ].space_ID.push_back( current_design.obtain_space( i ).ID ) ;
899         }
900
901
902         // create cluster for y = max
903         double max_depth = 0 ;
904         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
905         {
906             if ( current_design.obtain_space( i ).y + current_design.obtain_space( i ).depth >
                max_depth )
907                 max_depth = current_design.obtain_space( i ).y + current_design.obtain_space( i
                    ).depth ;
908         }
909
```

```
910             double max_y = max_depth ;
911             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
912             {
913                 if ( current_design.obtain_space( i ).y + current_design.obtain_space( i ).depth ==
                    max_depth && current_design.obtain_space( i ).y < max_y )
914                     max_y = current_design.obtain_space( i ).y ;
915             }
916
917             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
918             {
919                 if ( current_design.obtain_space( i ).y >= max_y )
920                     temp_groups[ 5 ].space_ID.push_back( current_design.obtain_space( i ).ID ) ;
921             }
922
923             // assign all spaces that do not belong to one of these groups to the last group
924             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
925             {
926                 bool space_found = false;
927                 for ( unsigned int j = 0 ; j < temp_groups.size() ; j++ )
928                 {
929                     for ( unsigned int k = 0 ; k < temp_groups[ j ].space_ID.size() ; k++ )
930                     {
931                         if ( current_design.obtain_space( i ).ID == temp_groups[ j ].space_ID[ k ] )
932                         {
933                             space_found = true ;
934                             break ;
935                         }
936                     }
937                     if ( space_found )
938                         break ;
939                 }
940
941                 if( !space_found )
942                 {
943                     temp_groups[ 6 ].space_ID.push_back( current_design.obtain_space( i ).ID ) ;
944                 }
945             }
946
947             groups = temp_groups ;
948         }
949
950         void Building_Performances::add_group( group_performance new_group )
951         {
952             groups.push_back( new_group );
953         }
954
955         void Building_Performances::delete_group(int group_index)
956         {
957             groups.erase(groups.begin() + group_index ) ;
958         }
959
960         int Building Performances::id count()
961         {
962             int id count = 0;
963             for(unsigned int i = 0 ; i < groups.size() ; i++ )
964             {
965                 id count += groups[i].space ID.size();
966             }
967
968             return id_count;
969         }
970
971         int Building_Performances::id_belonging_group(int& id)
972         {
973
974             for ( unsigned int i = 0 ; i < groups.size() ; i++ )
975             {
976                 for (unsigned int j = 0 ; j < groups[i].space_ID.size() ; j++ )
977                 {
978                     if ( groups[i].space_ID[j] == id)
979                     {
980                         return i;
981                     }
982                 }
983             }
984             // if no space objects can be found with the id give error and exit
985             std::cout<< "Error no space_ID :" << id <<" cannot be found, exiting now... (
                Building_Performances.hpp)"<<std::endl;
986             exit(1);
987         }
988
989         int Building_Performances::best_group_initial()
990         {
991             int best_index = 0;
```

```
992             double best_performance = groups[0].initial_performance[0];
993
994             for ( unsigned int i = 0 ; i < groups.size() ; i++ )
995             {
996                 for ( unsigned int j = 0 ; j < groups[i].initial_performance.size() ; j++ )
997                 {
998                     if( groups[i].initial_performance[j] > best_performance )
999                     {
1000                        best_performance = groups[i].initial_performance[j];
1001                        best_index = i;
1002                    }
1003                }
1004            }
1005            return best_index;
1006        } // best_initial_group
1007
1008        int Building_Performances::best_group_modified()
1009        {
1010           int best_index = 0;
1011
1012           double best_performance = groups[0].modified_performance[0];
1013
1014            for ( unsigned int i = 0 ; i < groups.size() ; i++ )
1015            {
1016                for ( unsigned int j = 0 ; j < groups[i].modified_performance.size() ; j++ )
1017                {
1018
1019                   if ( groups[i].modified_performance[j] > best_performance )
1020                   {
1021                       best_performance = groups[i].modified_performance[j];
1022                       best_index = i;
1023                   }
1024               }
1025           }
1026           return best_index;
1027       } // best_group_modified
1028
1029       int Building_Performances::worst_group_initial()
1030       {
1031           int worst_index = 0;
1032           double worst_performance = groups[0].initial_performance[0];
1033
1034           for ( unsigned int i = 0 ; i < groups.size()  ; i++)
1035           {
1036               for ( unsigned int j = 0 ; j < groups[i].initial_performance.size() ; j++ )
1037               {
1038                   if ( groups[i].initial_performance[j] < worst_performance )
1039                   {
1040                       worst_performance = groups[i].initial_performance[j];
1041                       worst_index = i;
1042                   }
1043               }
1044           }
1045           return worst index;
1046       } // worst group initial
1047
1048       int Building Performances::worst group modified()
1049       {
1050           int worst_index = 0;
1051           double worst_performance = groups[0].initial_performance[0];
1052
1053           for ( unsigned int i = 0 ; i < groups.size() ; i++ )
1054           {
1055               for (unsigned int j = 0 ; j < groups[i].modified_performance.size() ; j++ )
1056               {
1057                   if ( groups[i].modified_performance[j] < worst_performance )
1058                   {
1059                       worst_performance = groups[i].modified_performance[j];
1060                       worst_index = i;
1061                   }
1062               }
1063           }
1064           return worst_index;
1065       } // worst_group_modified
1066
1067       int Building_Performances::best_group_distance( )
1068       {
1069           int best_group = 0 ;
1070           double best_performance = 0;
1071           for ( unsigned int i = 0 ; i < groups[ 0 ].modified_performance.size() ; i++ )
1072           {
1073               best_performance += ( 1 - groups[ 0 ].modified_performance[ i ] ) ;
1074           }
1075           best_performance = pow( best_performance, 1.0 / groups[ 0 ].modified_performance.size()
```

```
            ) ;
1076
1077            for ( unsigned int i = 1 ; i < groups.size() ; i++ )
1078            {
1079                double temp_performance = 0 ;
1080
1081                for ( unsigned int j = 0 ; j < groups[ i ].modified_performance.size() ; j++ )
1082                {
1083                    temp_performance += ( 1 - groups[ i ].modified_performance[ j ] ) ;
1084                }
1085                temp_performance = pow( temp_performance, 1.0 / groups[ i
                    ].modified_performance.size() ) ;
1086
1087                if ( temp_performance < best_performance )
1088                {
1089                    best_performance = temp_performance ;
1090                    best_group = i ;
1091                }
1092            }
1093            return best_group ;
1094        } // best_group_distance
1095
1096        int Building_Performances::worst_group_distance( )
1097        {
1098            int worst_group = 0 ;
1099            double worst_performance = 0 ;
1100            for ( unsigned int i = 0 ; i < groups[ 0 ].modified_performance.size() ; i++ )
1101            {
1102                worst_performance += groups[ 0 ].modified_performance[ i ] ;
1103            }
1104            worst_performance = pow( worst_performance, 1.0 / groups[ 0
                ].modified_performance.size() ) ;
1105
1106            for ( unsigned int i = 0 ; i < groups.size() ; i++ )
1107            {
1108                double temp_performance = 0 ;
1109                for ( unsigned int j = 0 ; j < groups[ i ].modified_performance.size() ; j++ )
1110                {
1111                    temp_performance += groups[ i ].modified_performance[ j ] ;
1112                }
1113                temp_performance = pow( temp_performance, 1.0 / groups[ i
                    ].modified_performance.size() ) ;
1114
1115                if ( temp_performance < worst_performance )
1116                {
1117                    worst_performance = temp_performance ;
1118                    worst_group = i ;
1119                }
1120            }
1121            return worst_group ;
1122        } // worst group distance
1123
1124        int Building Performances::get initial space count()
1125        {
1126            return initial space count;
1127        }
1128
1129        double Building Performances::get initial volume()
1130        {
1131            return initial_volume;
1132        }
1133
1134    } // namespace Performance_Evaluation
1135    } // namespace HBO
1136    } // namespace BSO
1137
1138    #endif // BUILDING_PERFORMANCES
1139
```

```cpp
#ifndef PERFORMANCE_EVALUATION_HPP
#define PERFORMANCE_EVALUATION_HPP

#include <BSO/HBO/HBO_Settings.hpp>
#include <BSO/HBO/Performance_Evaluation/Non_Aggregate_Performances.hpp>
#include <BSO/HBO/Performance_Evaluation/Aggregate_Performances.hpp>

#include <iostream>
#include <vector>

namespace BSO { namespace HBO { namespace Performance_Evaluation {

    HBO::Performance_Evaluation::Building_Performances evaluate_performance(
    HBO::Performance_Evaluation::Building_Performances& build_perform, HBO::Settings& settings )
    {
        switch( settings.assessment_options )
        {
        case BSO::HBO::performance_assessment::INDIVIDUAL: { return
        BSO::HBO::Performance_Evaluation::non_aggregate_performance( build_perform, settings) ; }
        case BSO::HBO::performance_assessment::AGGREGATE: { return
        BSO::HBO::Performance_Evaluation::aggregate_performances( build_perform, settings) ;}
        case BSO::HBO::performance_assessment::NON_AGGREGATE: { return
        BSO::HBO::Performance_Evaluation::non_aggregate_performance( build_perform, settings ) ;}
        case BSO::HBO::performance_assessment::ARG_COUNT: { std::cout << "Error ARG_COUNT
        reached, exiting now... (Performance_Evaluation.hpp" << std::endl ; exit(1) ; }
        default: { std::cout << "Error default, exiting now... (Performance_Evaluation.hpp)" <<
        std::endl ; exit(1) ; }
        }
    }
} // Performance_Evaluation
} // HBO
} // BSO

#endif // PERFORMANCE_EVALUATION_HPP
```

```cpp
#ifndef NON_AGGREGATED_PERFORMANCES_HPP
#define NON_AGGREGATED_PERFORMANCES_HPP

#include <BSO/HBO/HBO_Settings.hpp>
#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>

#include <vector>

namespace BSO { namespace HBO { namespace Performance_Evaluation {

    /*
        Functions that do not aggregate space performances
    */

    HBO::Performance_Evaluation::Building_Performances non_aggregate_performance (
    HBO::Performance_Evaluation::Building_Performances& build_perform, HBO::Settings& settings)
    {
        HBO::Performance_Evaluation::Building_Performances temp_build_perform = build_perform;

        if ( settings.weights.empty() ) // if no weights are assigned, set all weights equal
        {
            for ( unsigned int i = 0 ; i <
            temp_build_perform.groups[0].initial_performance.size() ; i++ )
            {
                settings.weights.push_back( 1.0 /
                temp_build_perform.groups[0].initial_performance.size() );
            }
        }

        if ( settings.weights.size() != temp_build_perform.groups[0].initial_performance.size() )
        {
            std::cout<< "Amount of weights and initial performances do not match, exiting now...
            (Non_Aggregate_Performances.hpp)"<<std::endl; exit(1);
        }

        for ( unsigned int i = 0 ; i < temp_build_perform.groups.size() ; i++ )
        {
            for ( unsigned int j = 0 ; j <
            temp_build_perform.groups[i].initial_performance.size() ; j++ )
            {
                temp_build_perform.groups[i].modified_performance.push_back(
                temp_build_perform.groups[i].initial_performance[j] * settings.weights[j] );
            }
        }

        return temp_build_perform;
    }


} // namespace Performance_Evaluation
} // namespace HBO
} // namespace BSO

#endif // NON_AGGREGATED_PERFORMANCES_HPP
```

```cpp
#ifndef AGGREGATE_DISCIPLINES_HPP
#define AGGREGATE_DISCIPLINES_HPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/Structural_Design/SD_Results.hpp>
#include <BSO/Building_Physics/BP_Results.hpp>
#include <BSO/Trim_And_Cast.hpp>

#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/HBO_Settings.hpp>


#include <iostream>
#include <vector>
#include <cmath>

namespace BSO { namespace HBO { namespace Performance_Evaluation {

        /*
        Functions that aggregate space performances
        */

        Performance_Evaluation::Building_Performances
        aggregate_performances(Performance_Evaluation::Building_Performances& build_perform,
        Settings& settings)
        {
            int space_count;

            Performance_Evaluation::Building_Performances temp_build_perform = build_perform ;

            if( settings.weights.empty() ) // if no weights are initialized, give all
            disciplines equal weight
            {
                for ( unsigned int i = 0 ; i < temp_build_perform.groups[ 0
                ].initial_performance.size() ; i++ )
                {
                    settings.weights.push_back( 1.0 / temp_build_perform.groups[ 0
                    ].initial_performance.size() ) ;
                }
            }

            if ( settings.weights.size() != temp_build_perform.groups[ 0
            ].initial_performance.size() ) // check whether there are enough weights for the
            performances
            {
                std::cout << "The amount of weights does not equal the amount of performances,
                exiting now... (Aggregate_Performances.hpp)" << std::endl ; exit( 1 ) ;
            }

            for (unsigned int i = 0 ; i < temp_build_perform.groups.size() ; i++)
            {
                switch(settings.aggregate options)
                {
                case aggregate disciplines::SUMMATION:
                    for(unsigned int j = 0 ; j < temp build perform.groups[ i
                    ].initial performance.size() ; j++ )
                    {
                        if( j == 0 )
                        {
                            temp_build_perform.groups[ i
                            ].modified_performance.push_back(temp_build_perform.groups[ i
                            ].initial_performance[ j ] * settings.weights[ j ] ) ;
                        }
                        else if( j != 0 )
                        {
                            temp_build_perform.groups[ i ].modified_performance[ 0 ] +=
                            temp_build_perform.groups[ i ].initial_performance[ j ] *
                            settings.weights[ j ] ;
                        }
                        else{ std::cout<<"Error in summation, exiting now...
                        (Aggregate_Performances.hpp)";  exit( 1 );  }
                    }
                    break;

                case aggregate_disciplines::PRODUCT:
                    for( unsigned int j = 0 ; j < temp_build_perform.groups[ i
                    ].initial_performance.size() ; j++ )
                    {
                        if( j == 0 )
                        {
                            temp_build_perform.groups[ i
                            ].modified_performance.push_back(temp_build_perform.groups[ i
                            ].initial_performance[ j ] * settings.weights[ j ] ) ;
                        }
```

```
68                          else if( j != 0 )
69                          {
70                              temp_build_perform.groups[ i ].modified_performance[ 0 ] *=
                                temp_build_perform.groups[ i ].initial_performance[ j ] *
                                settings.weights[ j ] ;
71                          }
72                          else { std::cout << "Error in product, exiting now...
                            (Aggregate_Performances.hpp)" << std::endl ; exit( 1 ) ; }
73                      }
74                      break;
75
76                  case aggregate_disciplines::DISTANCE:
77                  {
78
79                      for ( unsigned int j = 0 ; j < temp_build_perform.groups[ i
                        ].initial_performance.size() ; j++ )
80                      {
81                          if ( j == 0 )
82                          {
83                              temp_build_perform.groups[ i ].modified_performance.push_back(pow( (
                                1 - temp_build_perform.groups[ i ].initial_performance[ j ] *
                                settings.weights[ j ] ), 2) ) ;
84                          }
85                          else if( j != 0 )
86                          {
87                              temp_build_perform.groups[ i ].modified_performance[ 0 ] += pow( ( 1
                                - temp_build_perform.groups[ i ].initial_performance[ j ] *
                                settings.weights[ j ] ), 2 )  ;
88                          }
89                          else { std::cout << "Error in euclidean distance, exiting now... (
                            Aggregate_Performances.hpp)" << std::endl ; exit(1) ; }
90                      }
91                      temp_build_perform.groups[ i ].modified_performance[ 0 ] = pow(
                        temp_build_perform.groups[ i ].modified_performance[ 0 ], 1.0 /
                        temp_build_perform.groups[ i ].initial_performance.size() ) ;
92                      temp_build_perform.groups[ i ].modified_performance[ 0 ] = pow( 2, 1.0/2 ) -
                        temp_build_perform.groups[ i ].modified_performance[ 0 ] ;
93
94                      break ;
95                  }
96                  case aggregate_disciplines::INVERSE_PRODUCT:
97                      for ( unsigned int j = 0 ; j < temp_build_perform.groups[ i
                        ].initial_performance.size() ; j++ )
98                      {
99                          if ( j == 0 )
100                         {
101                             temp_build_perform.groups[ i ].modified_performance.push_back( (1 -
                                temp_build_perform.groups[ i ].initial_performance[ j ] ) *
                                settings.weights[ j ] );
102                         }
103                         else if ( j != 0 )
104                         {
105                             temp build perform.groups[ i ].modified performance[ 0 ] *= ( 1 -
                                temp build perform.groups[ i ].initial performance[ j ] ) *
                                settings.weights[ j ];
106                         }
107                         else { std::cout << "Error in inverse product, exiting now... (
                            Aggregate Performances.hpp)" << std::endl ; exit( 1 ); }
108                     }
109                     break;
110                 case aggregate_disciplines::ARG_COUNT:
111                     std::cout << "Error ARG_COUNT in aggregate options, exiting now...
                        (Aggregate_Performances.hpp)" << std::endl ;
112                     exit( 1 ) ;
113                     break ;
114                 default:
115                     std::cout << "Error in aggregate options, exiting now...
                        (Aggregate_Performances.hpp)" << std::endl ;
116                     exit( 1 ) ;
117                 } // end switch
118
119             } // for all temp_build.groups
120
121             return temp_build_perform ;
122
123         } // aggregate_performances
124
125
126 }   // namespace Performance_evaluation
127 }   // namespace HBO
128 } // namespace BSO
129
130 #endif // AGGREGATE_DISCIPLINES
131
```

```cpp
#ifndef SPACE_RANKING_NON_AGGREGATED_HPP
#define SPACE_RANKING_NON_AGGREGATED_HPP

#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/HBO_Settings.hpp>

#include <vector>
#include <cmath>

namespace BSO { namespace HBO { namespace Space_Selection {

    /*
        Functions to rank space with non aggregated performances
        ALL LISTS ARE GENERATED WITH WORST PERFORMANCE AT THE ZERO INDEX
    */
    std::vector<int> rank_spaces_best_performance(
    Performance_Evaluation::Building_Performances& , HBO::Settings& ) ;
    std::vector<int> rank_spaces_worst_performance(
    Performance_Evaluation::Building_Performances& , HBO::Settings& ) ;
    std::vector<int> rank_spaces_best_distance( Performance_Evaluation::Building_Performances&,
    HBO::Settings& ) ;
    std::vector<int> rank_spaces_worst_distance( Performance_Evaluation::Building_Performances&,
    HBO::Settings& ) ;
    std::vector<int> rank_spaces_alter_best( Performance_Evaluation::Building_Performances& ,
    HBO::Settings& ) ;
    std::vector<int> rank_spaces_alter_worst( Performance_Evaluation::Building_Performances& ,
    HBO::Settings& ) ;

    std::vector<int> rank_spaces( Performance_Evaluation::Building_Performances& build_perform,
    HBO::Settings& settings )
    {
        std::vector<int> spaces_ranked ;

        switch( settings.ranking_type)
        {
            case space_ranking::BEST: { spaces_ranked = rank_spaces_best_performance(
            build_perform, settings ) ; break ; }
            case space_ranking::WORST: { spaces_ranked = rank_spaces_worst_performance(
            build_perform, settings ) ; break ; }
            case space_ranking::BEST_DISTANCE: { spaces_ranked = rank_spaces_best_distance(
            build_perform, settings ) ; break ; }
            case space_ranking::WORST_DISTANCE: { spaces_ranked = rank_spaces_worst_distance(
            build_perform, settings ) ; break ; }
            case space_ranking::ALTER_BEST: { spaces_ranked = rank_spaces_alter_best(
            build_perform, settings ) ; break ; }
            case space_ranking::ALTER_WORST: { spaces_ranked = rank_spaces_worst_performance(
            build_perform, settings ) ; break ;}
            case space_ranking::ARG_COUNT: { std::cout << "Error ARG_COUNT, exiting now...
            (Space_Ranking.hpp)" << std::endl ; exit(1) ; }
            default: { std::cout << "Error default, exiting now ... (Space_Ranking.hpp)" <<
            std::endl; exit(1) ; }
        }

        return spaces ranked;
    }


    std::vector<int> rank spaces best performance(
    Performance_Evaluation::Building_Performances& build_perform, HBO::Settings& settings )
    {
        // Orders the spaces by their best performance, no matter what the other performances
        are
        BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform =
        build_perform;
        std::vector<int> spaces_ranked;
        // take the best group at a time and add its IDs to the list
        for ( unsigned int i = 0 ; i < build_perform.groups.size() ; i++ )
        {
            int best_group = temp_build_perform.best_group_modified();

            for ( unsigned int j = 0 ; j < temp_build_perform.groups[best_group].space_ID.size()
            ; j++ ) // This method ranks the best space at element 0, so it needs to be reversed
            {
                spaces_ranked.push_back(temp_build_perform.groups[best_group].space_ID[j]);
            }
            temp_build_perform.groups.erase(temp_build_perform.groups.begin() + best_group);
        }
        // reverse the space list
        for ( unsigned int i = 0 ; i < (spaces_ranked.size() / 2) ; i++ )
        {
            int temp_id = spaces_ranked[i];
            spaces_ranked[i] = spaces_ranked[ spaces_ranked.size() - i - 1 ];
            spaces_ranked[spaces_ranked.size() - i - 1 ] = temp_id;
        }
```

```
66          return spaces_ranked;
67      }
68
69
70      std::vector<int> rank_spaces_worst_performance(
        Performance_Evaluation::Building_Performances& build_perform, HBO::Settings& settings)
71      {
72          // Order spaces by their worst performance, not matter what the other performances are
73          BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform =
            build_perform;
74          std::vector<int> spaces_ranked;
75
76          //take the worst group at a time and add its IDs to the list
77          for ( unsigned int i = 0 ; i < build_perform.groups.size() ; i++ )
78          {
79              int worst_group = temp_build_perform.worst_group_modified();
80
81              for ( unsigned int j = 0 ; j <
                temp_build_perform.groups[worst_group].space_ID.size() ; j++ )
82              {
83                  spaces_ranked.push_back(temp_build_perform.groups[worst_group].space_ID[j]);
84              }
85              temp_build_perform.groups.erase(temp_build_perform.groups.begin() + worst_group);
86          }
87
88          return spaces_ranked;
89      }
90
91      std::vector< int > rank_spaces_best_distance( Performance_Evaluation::Building_Performances&
        build_perform, HBO::Settings& settings )
92      {
93          // Order spaces by the distance to the utopia point ( 1, 1)
94
95          BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform =
            build_perform ;
96          std::vector< int > spaces_ranked ;
97
98          // take the group with the shortest distance and at its IDs to the list
99          for ( unsigned int i = 0 ; i < build_perform.groups.size() ; i++ )
100         {
101             int best_group = temp_build_perform.best_group_distance();
102
103             for ( unsigned int j = 0 ; j < temp_build_perform.groups[ best_group
                ].space_ID.size() ; j++ )
104             {
105                 spaces_ranked.push_back( temp_build_perform.groups[ best_group ].space_ID[ j ] ) ;
106             }
107             temp_build_perform.groups.erase( temp_build_perform.groups.begin() + best_group ) ;
108         }
109
110         // reverse the space list so the space with the largest distance is at index 0
111         for ( unsigned int i = 0 ; i < (spaces ranked.size() / 2) ; i++ )
112         {
113             int temp id = spaces ranked[i];
114             spaces ranked[i] = spaces ranked[ spaces ranked.size() - i - 1 ];
115             spaces ranked[spaces ranked.size() - i - 1 ] = temp id;
116         }
117         return spaces ranked ;
118     }
119
120     std::vector< int > rank_spaces_worst_distance(
        Performance_Evaluation::Building_Performances& build_perform, HBO::Settings& settings )
121     {
122         // Order spaces by the distance to ( 0, 0)
123         BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform =
            build_perform ;
124         std::vector< int > spaces_ranked ;
125
126         // take the group with the shortest distance and at its IDs to the list
127         for ( unsigned i = 0 ; i < build_perform.groups.size() ; i++ )
128         {
129             int worst_group = temp_build_perform.worst_group_distance() ;
130
131             for ( unsigned int j = 0 ; j < temp_build_perform.groups[ worst_group
                ].space_ID.size() ; j++ )
132             {
133                 spaces_ranked.push_back( temp_build_perform.groups[ worst_group ].space_ID[ j ]
                    ) ;
134             }
135             temp_build_perform.groups.erase( temp_build_perform.groups.begin() + worst_group ) ;
136         }
137         return spaces_ranked ;
138     }
139
```

```
140     std::vector<int> rank_spaces_alter_best( Performance_Evaluation::Building_Performances&
        build_perform, HBO::Settings& settings)
141     {
142         // Order spaces alternating between the disciplines, choosing the best performance
143         BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform =
            build_perform;
144         std::vector<int> spaces_ranked;
145
146         int discipline_number = 0 ;
147         for ( unsigned int i = 0 ; i < build_perform.groups.size() ; i++ )
148         {
149             discipline_number++ ;
150             if ( discipline_number >= build_perform.groups[ 0 ].modified_performance.size() )
151                 discipline_number = 1 ;
152
153             double best_performance = temp_build_perform.groups[ 0 ].modified_performance[
                discipline_number ] ;
154             int best_group = 0 ;
155             for ( unsigned int j = 0 ; j < temp_build_perform.groups.size() ; j++ )
156             {
157                 if ( temp_build_perform.groups[ j ].modified_performance[ discipline_number ] >
                    best_performance )
158                 {
159                     best_performance = temp_build_perform.groups[ j ].modified_performance[
                        discipline_number ] ;
160                     best_group = j ;
161                 }
162             }
163
164             for ( unsigned int j = 0 ; j < temp_build_perform.groups[ best_group
                ].space_ID.size() ; j++ )
165             {
166                 spaces_ranked.push_back( temp_build_perform.groups[ best_group ].space_ID[ j ] ) ;
167             }
168             temp_build_perform.delete_group( best_group ) ;
169         }
170
171         for ( unsigned int i = 0 ; i < ( spaces_ranked.size() / 2 ) ; i++ )
172         {
173             int temp_id = spaces_ranked[i];
174             spaces_ranked[i] = spaces_ranked[spaces_ranked.size() - i - 1 ];
175             spaces_ranked[spaces_ranked.size() - i - 1 ] = temp_id;
176         }
177         return spaces_ranked;
178
179     }
180
181     std::vector<int> rank_spaces_alter_worst( Performance_Evaluation::Building_Performances&
        build_perform, HBO::Settings& settings)
182     {
183         // Order spaces alternating between the disciplines, choosing the worst performace
184
185         BSO::HBO::Performance Evaluation::Building Performances temp build perform =
            build perform;
186         std::vector<int> spaces ranked;
187
188         for ( unsigned int i = 0 ; i < build perform.groups.size() ; i++ )
189         {
190             for ( unsigned int j = 0 ; j < build_perform.groups[0].modified_performance.size() ;
                j++ )
191             {
192                 double worst_perform = temp_build_perform.groups[0].modified_performance[j];
193                 int worst_group = 0;
194
195                 for ( unsigned int k = 0 ; k < temp_build_perform.groups.size() ; k++ )
196                 {
197                     if ( temp_build_perform.groups[k].modified_performance[j] < worst_perform )
198                     {
199                         worst_group = k;
200                         worst_perform = temp_build_perform.groups[k].modified_performance[j];
201                     }
202                 }
203
204                 for ( unsigned int k = 0 ; k <
                    temp_build_perform.groups[worst_group].space_ID.size() ; k++ )
205                 {
206                     spaces_ranked.push_back(temp_build_perform.groups[worst_group].space_ID[k]);
207                 }
208                 temp_build_perform.groups.erase(temp_build_perform.groups.begin() + worst_group );
209             }
210         }
211
212         return spaces_ranked;
213     }
```

```
214
215   } // namespace Space_Selection
216   } // namespace HBO
217   } // namespace BSO
218
219   #endif // SPACE_RANKING_NON_AGGREGATED_HPP
```

Building Modification

```cpp
#ifndef BUILDING_MODIFICATION_HPP
#define BUILDING_MODIFICATION_HPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/Building_Modification/Rescaling.hpp>
#include <BSO/HBO/Building_Modification/Sweeping.hpp>

#include <BSO/HBO/HBO_Settings.hpp>

#include <iostream>
#include <vector>

namespace BSO { namespace HBO { namespace Building_Modification {

    BSO::Spatial_Design::MS_Building building_modification( BSO::Spatial_Design::MS_Building&
    current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    BSO::HBO::Settings& settings )
    {

        switch( settings.modification_options )
        {
            case BSO::HBO::building_modification::SCALE: { return
            BSO::HBO::Building_Modification::rescale_building( current_design, build_perform,
            settings ) ; }
            case BSO::HBO::building_modification::SWEEP: { return
            BSO::HBO::Building_Modification::sweep_building( current_design, build_perform,
            settings ) ; }
            case BSO::HBO::building_modification::ARG_COUNT: { std::cout <<" Error ARG_COUNT,
            exiting now... ( Building_Modification.hpp ) " << std::endl; exit(1) ; }
            default: { std::cout << "Error default. Exiting now... ( Building_Modification.hpp )
            " << std::endl ; exit(1) ;}
        } // end switch

    } // building_modification()
} // Building_Modification
} // HBO
} // BSO

    #endif // BUILDING_MODIFICATION_HPP
```

```cpp
#ifndef SPACE_REMOVAL_HPP
#define SPACE_REMOVAL_HPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/Trim_And_Cast.hpp>
#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/HBO_Settings.hpp>

#include <AEI Grammar/Grammar_2.hpp>

#include <iostream>
#include <vector>
#include <cmath>

namespace BSO { namespace HBO { namespace Space_Selection {

    /*
        Functions to remove spaces from building spatial designs
    */
        BSO::Spatial_Design::MS_Building remove_1_space( BSO::Spatial_Design::MS_Building&
        current_design,  BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
        std::vector<int>& spaces_ranked ) ;
        BSO::Spatial_Design::MS_Building remove_x_worst_spaces(
        BSO::Spatial_Design::MS_Building& current_design, std::vector<int>& spaces_ranked,
        BSO::HBO::Settings& settings ) ;
        BSO::Spatial_Design::MS_Building remove_x_best_spaces( BSO::Spatial_Design::MS_Building&
        current_design, std::vector<int>& spaces_ranked, BSO::HBO::Settings& settings ) ;
        BSO::Spatial_Design::MS_Building remove_top_floor( BSO::Spatial_Design::MS_Building&
        current_design, std::vector<int>& spaces_ranked, BSO::HBO::Settings& settings ) ;
        BSO::Spatial_Design::MS_Building remove_bp_function( BSO::Spatial_Design::MS_Building&
        current_design, BSO::HBO::Settings& settings ) ;

    BSO::Spatial_Design::MS_Building removal_of_spaces( BSO::Spatial_Design::MS_Building&
    current_design, std::vector<int>& spaces_ranked ,
    BSO::HBO::Performance_Evaluation::Building_Performances& build_perform, BSO::HBO::Settings&
    settings )
    {
        switch(settings.removal_type)
        {
            case HBO::space_removal_type::ONE_WORST: { return remove_1_space( current_design,
            build_perform, spaces_ranked ) ; break ; }
            //case HBO::space_removal_type::TENTH: { settings.space_removal_requested = ceil(
            current_design.obtain_space_count() * 0.10 ) ; return remove_x_worst_spaces(
            current_design, spaces_ranked, settings) ; }
            //case HBO::space_removal_type::TWENTIETH: { settings.space_removal_requested =
            ceil( current_design.obtain_space_count() * 0.20 ) ; return remove_x_worst_spaces(
            current_design, spaces_ranked, settings) ; }
            case HBO::space_removal_type::THIRTIETH: { settings.space_removal_requested = ceil(
            current_design.obtain_space_count() * 0.30 ) ; return remove_x_worst_spaces(
            current_design, spaces_ranked, settings) ; }
            case HBO::space removal type::FORTIETH: { settings.space removal requested = ceil(
            current design.obtain space count() * 0.40 ) ; return remove x worst spaces(
            current design, spaces ranked, settings ) ; }
            case HBO::space removal type::FIFTIETH: { settings.space removal requested = ceil(
            current design.obtain space count() * 0.50 ) ; return remove x worst spaces(
            current design, spaces ranked, settings ) ; }
            //case HBO::space removal type::X BEST: { return remove x best spaces(
            current design, spaces ranked, settings ) ; break ; }
            //case HBO::space_removal_type::TOP_FLOOR: { return remove_top_floor(
            current_design, spaces_ranked, settings ) ; break ; }
            //case HBO::space_removal_type::BP_REMOVAL: { return remove_bp_function(
            current_design, settings ); break ; }
            case HBO::space_removal_type::ARG_COUNT: { std::cout<<"Error ARG_COUNT reached,
            exiting now... ( Space_Removal.hpp)"<<std::endl; exit(1) ; }
            default: { std::cout<<"Error default, exiting now...
            (Space_Removal.hpp)"<<std::endl; exit(1) ; }
        }
    } // removal_of_spaces

    BSO::Spatial_Design::MS_Building remove_1_space( BSO::Spatial_Design::MS_Building&
    current_design,  BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    std::vector<int>& spaces_ranked )
    {
        // removes the worst spaces in the building design until spaces are removed equal to the
        worst group in build_perform
        int remove = build_perform.groups[ build_perform.worst_group_modified()
        ].space_ID.size() ;
        BSO::Spatial_Design::MS_Building new_design = current_design ;
        for ( unsigned int i = 0 ; i < remove ; i++ )
        {
            new_design.delete_space(new_design.get_space_index(spaces_ranked[ i ]));
        }

        return new_design;
```

```
55      } // remove_1_space
56
57      BSO::Spatial_Design::MS_Building remove_x_worst_spaces( BSO::Spatial_Design::MS_Building&
        current_design, std::vector<int>& spaces_ranked, BSO::HBO::Settings& settings )
58      {
59          // removes the x worst spaces in the building design
60          BSO::Spatial_Design::MS_Building new_design = current_design;
61
62          for ( unsigned int i = 0 ; i < settings.space_removal_requested ; i++ )
63          {
64              new_design.delete_space( new_design.get_space_index(spaces_ranked[ i ]));
65          }
66
67          return new_design;
68      } // remove_x_worst_spaces
69
70      BSO::Spatial_Design::MS_Building remove_x_best_spaces( BSO::Spatial_Design::MS_Building&
        current_design, std::vector<int>& spaces_ranked, BSO::HBO::Settings& settings )
71      {
72          // removes the x best spaces in the building design
73          BSO::Spatial_Design::MS_Building new_design = current_design;
74
75          for ( unsigned int i = 0 ; i < settings.space_removal_requested ; i++ )
76          {
77              new_design.delete_space(
                new_design.get_space_index(spaces_ranked[spaces_ranked.size() - i - 1] ));
78          }
79
80          return new_design;
81      } // remove_x_best_spaces
82
83      BSO::Spatial_Design::MS_Building remove_top_floor( BSO::Spatial_Design::MS_Building&
        current_design, std::vector<int>& spaces_ranked, BSO::HBO::Settings& settings )
84      {
85          // removes the top floor of a building spatial design
86          BSO::Spatial_Design::MS_Building new_design = current_design;
87
88          // reset the removal count
89          settings.space_removal_requested = 0;
90
91          // find the highest z value
92          double z_high = 0 ;
93          for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
94          {
95              if ( new_design.obtain_space( i ).z > z_high )
96              {
97                  z_high = new_design.obtain_space( i ).z ;
98              }
99          }
100
101         // remove all spaces with the highest z value
102         for ( unsigned int i = 0 ; i < new design.obtain space count() ; i++ )
103         {
104             if ( new design.obtain space( i ).z == z high )
105             {
106                 settings.space removal requested++ ;
107                 new design.delete space( i ) ;
108                 i-- ;
109             }
110         }
111
112         return new_design;
113     } // remove_1_floor
114
115     BSO::Spatial_Design::MS_Building remove_bp_function( BSO::Spatial_Design::MS_Building&
        current_design, BSO::HBO::Settings& settings )
116     {
117         BSO::Spatial_Design::MS_Building new_design = current_design ;
118         std::vector< int > space_indexes_for_removal ;
119         settings.space_removal_requested = 0 ;
120
121         BSO::Spatial_Design::MS_Conformal CF( new_design, &(BSO::Grammar::grammar_2) ) ;
122
123         for ( unsigned int i = 0 ; i < CF.get_space_count() ; i++ )
124         {
125             int adjacent_spaces = 0 ;
126             for ( unsigned int j = 0 ; j < 6 ;j++ )
127             {
128                 bool space_found = false;
129
130                 for ( unsigned int k = 0 ; k < CF.get_space( i )->get_surface_ptr( j
                    )->get_rectangle_count() ; k++ )
131                 {
132                     BSO::Spatial_Design::Geometry::Rectangle* rectangle =
```

```
                        CF.get_space(i)->get_surface_ptr(j)->get_rectangle_ptr(k) ;
133
134                     for ( unsigned int l = 0 ; l < rectangle->get_surface_count() ; l++ )
135                     {
136                         if ( rectangle->get_surface_count() == 2 )
137                         {
138                             space_found = true ;
139                             break ;
140                         }
141                     }
142
143                     if ( space_found )
144                         break ;
145                 }
146
147                 adjacent_spaces++ ;
148             }
149
150             if ( adjacent_spaces <= 2 )
151             {
152                 new_design.delete_space( new_design.get_space_index( CF.get_space( i )->get_ID()
                    ) ) ;
153                 settings.space_removal_requested++ ;
154             }
155         }
156
157         return new_design ;
158     } // remove_bp_function
159
160 } // Space_Selection
161 } // HBO
162 } // BSO
163
164 #endif // SPACE_REMOVAL_HPP
```

```cpp
#ifndef RESCALING_HPP
#define RESCALING_HPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/HBO_Settings.hpp>

#include <vector>
#include <cmath>

namespace BSO { namespace HBO { namespace Building_Modification {

    BSO::Spatial_Design::MS_Building rescale_building( BSO::Spatial_Design::MS_Building&
    current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    BSO::HBO::Settings& settings )
    {
        BSO::Spatial_Design::MS_Building new_design = current_design ;

        if ( new_design.get_volume() == build_perform.get_initial_volume() )
        {
            std::cout << "Warning, volume before scaling is equal to initial volume. " <<
            std::endl;
        }

        switch( settings.rescaling )
        {
            case rescaling_options::X:
            {
                new_design.scale_x( build_perform.get_initial_volume() / new_design.get_volume()
                ) ;
                return new_design;
            }
            case rescaling_options::Y:
            {
                new_design.scale_y( build_perform.get_initial_volume() / new_design.get_volume()
                ) ;
                return new_design;
            }
            case rescaling_options::Z:
            {
                new_design.scale_z( build_perform.get_initial_volume() / new_design.get_volume()
                ) ;
                return new_design;
            }
            case rescaling_options::XY:
            {
                double factor = sqrt( build_perform.get_initial_volume() /
                new_design.get_volume() ) ;
                new_design.scale_x( factor ) ;
                new_design.scale_y( factor ) ;
                return new design;
            }
            case rescaling options::XZ:
            {
                double factor = sqrt( build perform.get initial volume() /
                new design.get volume() ) ;
                new design.scale x( factor ) ;
                new design.scale z( factor ) ;
                return new_design ;
            }
            case rescaling_options::YZ:
            {
                double factor = sqrt( build_perform.get_initial_volume() /
                new_design.get_volume() ) ;
                new_design.scale_y( factor) ;
                new_design.scale_z( factor) ;
                return new_design ;
            }
            case rescaling_options::XYZ:
            {
                double factor = cbrt( build_perform.get_initial_volume() /
                new_design.get_volume() ) ;
                new_design.scale_x( factor ) ;
                new_design.scale_y( factor ) ;
                new_design.scale_z( factor ) ;
                return new_design ;
            }
            case rescaling_options::ARG_COUNT:
            {
                std::cout << "Error ARG_COUNT, exiting now... (Rescaling.hpp)" << std::endl ;
                exit(1) ;
            }
            default:
            {
```

```
74                  std::cout << "Error default, exiting now... (Rescaling.hpp)" << std::endl ;
                    exit(1) ;
75              }
76          } // end switch
77      } // rescale_building
78
79  } // namespace Building_Modification
80  } // namespace HBO
81  } // namespace BSO
82
83  #endif // RESCALING_HPP
```

```cpp
#ifndef SWEEPING_HPP
#define SWEEPING_HPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/HBO_Settings.hpp>

#include <vector>
#include <cmath>

namespace BSO { namespace HBO { namespace Building_Modification {

    /*
        Functions that sweep a building spatial design
    */

    BSO::Spatial_Design::MS_Building sweep_x_axis( BSO::Spatial_Design::MS_Building&,
    BSO::HBO::Performance_Evaluation::Building_Performances&, BSO::HBO::Settings& );
    BSO::Spatial_Design::MS_Building sweep_y_axis( BSO::Spatial_Design::MS_Building&,
    BSO::HBO::Performance_Evaluation::Building_Performances&, BSO::HBO::Settings& );
    BSO::Spatial_Design::MS_Building sweep_z_axis( BSO::Spatial_Design::MS_Building&,
    BSO::HBO::Performance_Evaluation::Building_Performances&, BSO::HBO::Settings& );
    /*
    BSO::Spatial_Design::MS_Building sweep_building( BSO::Spatial_Design::MS_Building&
    current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    BSO::HBO::Settings& settings )
    {
        switch (settings.sweep_option)
        {
            case HBO::sweeping_options::X: { BSO::Spatial_Design::MS_Building temp_design =
            sweep_x_axis( current_design, build_perform, settings ) ; return temp_design ; }
            case HBO::sweeping_options::Y: { BSO::Spatial_Design::MS_Building temp_design =
            sweep_y_axis( current_design, build_perform, settings ) ; return temp_design ; }
            case HBO::sweeping_options::Z: { BSO::Spatial_Design::MS_Building temp_design =
            sweep_z_axis( current_design, build_perform, settings ) ; return temp_design ; }
            case HBO::sweeping_options::ARG_COUNT: { std::cout<< "Error ARG_COUNT, exiting
            now... ( Sweeping.hpp)"<<std::endl; exit(1) ; }
            default: { std::cout<< "Error default, exiting now... (Sweeping.hpp)"<<std::endl;
            exit(1) ; }
        } // end switch
    } // sweep_building()
    */

    BSO::Spatial_Design::MS_Building sweep_building( BSO::Spatial_Design::MS_Building&
    current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    BSO::HBO::Settings& settings)
    {
        switch ( settings.sweep_option )
        {
            case HBO::sweeping_options::X_Z:
            {
                settings.sweep location = HBO::sweeping location::ZERO ;
                return sweep x axis( current design, build perform, settings ) ;
            }
            case HBO::sweeping options::X F:
            {
                settings.sweep location = HBO::sweeping location::FIFTY ;
                return sweep x axis( current design, build perform, settings ) ;
            }
            case HBO::sweeping_options::X_H:
            {
                settings.sweep_location = HBO::sweeping_location::HUNDRED ;
                return sweep_x_axis( current_design, build_perform, settings ) ;
            }
            case HBO::sweeping_options::Y_Z:
            {
                settings.sweep_location = HBO::sweeping_location::ZERO ;
                return sweep_y_axis( current_design, build_perform, settings ) ;
            }
            case HBO::sweeping_options::Y_F:
            {
                settings.sweep_location = HBO::sweeping_location::FIFTY ;
                return sweep_y_axis( current_design, build_perform, settings ) ;
            }
            case HBO::sweeping_options::Y_H:
            {
                settings.sweep_location = HBO::sweeping_location::HUNDRED ;
                return sweep_y_axis( current_design, build_perform, settings ) ;
            }
            case HBO::sweeping_options::Z_Z:
            {
                settings.sweep_location = HBO::sweeping_location::ZERO ;
                return sweep_z_axis( current_design, build_perform, settings ) ;
            }
```

```
73              case HBO::sweeping_options::Z_F:
74              {
75                  settings.sweep_location = HBO::sweeping_location::FIFTY ;
76                  return sweep_z_axis( current_design, build_perform, settings ) ;
77              }
78              case HBO::sweeping_options::Z_H:
79              {
80                  settings.sweep_location = HBO::sweeping_location::HUNDRED ;
81                  return sweep_z_axis( current_design, build_perform, settings ) ;
82              }
83              case HBO::sweeping_options::ARG_COUNT:
84              {
85                  std::cout<< "Error ARG_COUNT, exiting now... ( Sweeping.hpp)"<<std::endl;
86                  exit(1) ;
87              }
88              default:
89              {
90                  std::cout << "Error default, exiting now... ( Sweeping.hpp)" << std::endl ;
91              }
92          } // end switch ( sweep_options )
93      } // sweep_building
94
95      BSO::Spatial_Design::MS_Building sweep_x_axis( BSO::Spatial_Design::MS_Building&
        current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
        BSO::HBO::Settings& settings )
96      {
97          /*
98              Determine the values for sweeping
99          */
100
101         BSO::Spatial_Design::MS_Building new_design = current_design;
102
103         double area_swept = 0.0;
104         double x_value;
105
106         switch(settings.sweep_location)
107         {
108             case HBO::sweeping_location::ZERO: { x_value = 0; break;}
109             case HBO::sweeping_location::FIFTY:
110             {
111                 double temp_x = 0;
112                 for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
113                 {
114                     if ( new_design.obtain_space(i).x + new_design.obtain_space(i).width >
                        temp_x )
115                     {
116                         temp_x = new_design.obtain_space(i).x + new_design.obtain_space(i).width;
117                     }
118                 }
119                 x_value = temp_x * 0.5 ;
120                 break;
121             }
122             case HBO::sweeping location::HUNDRED:
123             {
124                 double temp x = 0;
125                 for ( unsigned int i = 0 ; i < new design.obtain space count() ; i++ )
126                 {
127                     if ( new design.obtain space(i).x + new design.obtain space(i).width >
                        temp_x )
128                     {
129                         temp_x = new_design.obtain_space(i).x + new_design.obtain_space(i).width ;
130                     }
131                 }
132                 x_value = temp_x;
133                 break;
134             }
135             case HBO::sweeping_location::ARG_COUNT:
136             {
137                 std::cout<<"Error ARG_COUNT, exiting now... (Sweeping.hpp)"<<std::endl; exit(1);
138                 break;
139             }
140             default:
141             {
142                 std::cout<<"Error default, exiting now... (Sweeping.hpp)"<<std::endl; exit(1);
143             }
144         }
145
146         // Check whether there are spaces at that x value_comp
147         bool space_at_x = false ;
148         for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
149         {
150             if ( new_design.obtain_space(i).x <= x_value && new_design.obtain_space(i).x +
                new_design.obtain_space(i).width > x_value )
151             {
```

```
152                 space_at_x = true ;
153                 break ;
154             }
155         }
156 
157         int range = 10 ;
158 
159         // If there is no space found at the x value, shift the value with small increments
            until a value is found for which a space is found
160         while ( !space_at_x )
161         {
162             if ( new_design.obtain_space_count() == 0 )
163             {
164                 settings.terminate_simulation = true ;
165                 return new_design ;
166             }
167             for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
168             {
169                 if ( new_design.obtain_space(i).x <= ( x_value - range ) &&
                    new_design.obtain_space(i).x + new_design.obtain_space(i).width > ( x_value -
                    range ) )
170                 {
171                     space_at_x = true ;
172                     x_value -= range ;
173                     break ;
174                 }
175                 else if ( new_design.obtain_space(i).x <= ( x_value + range ) &&
                    new_design.obtain_space(i).x + new_design.obtain_space(i).width > ( x_value +
                    range ) )
176                 {
177                     space_at_x = true ;
178                     x_value += range ;
179                     break ;
180                 }
181             }
182             range += 10 ;
183         } // end while() , note that no additional stopping criteria is formulated.
184 
185         // determine the area that is being swept
186         for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
187         {
188             if ( new_design.obtain_space(i).x <= x_value && new_design.obtain_space(i).x +
                new_design.obtain_space(i).width > x_value )
189             {
190                 area_swept += new_design.obtain_space(i).depth *
                    new_design.obtain_space(i).height ;
191             }
192         }
193 
194         // the length of sweeping
195         double sweeping distance = ( build perform.get initial volume() -
            new design.get volume() ) / area swept ;
196 
197         new design.sweep x( x value, sweeping distance );
198 
199         return new design;
200     } // sweep x axis()
201 
202     BSO::Spatial_Design::MS_Building sweep_y_axis( BSO::Spatial_Design::MS_Building&
        current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
        BSO::HBO::Settings& settings )
203     {
204         /*
205             Determines the values to sweep a building over the y axis
206         */
207 
208         BSO::Spatial_Design::MS_Building new_design = current_design ;
209 
210         double area_swept = 0 ;
211         double y_value;
212         double sweeping_distance ;
213 
214         switch(settings.sweep_location)
215         {
216             case HBO::sweeping_location::ZERO: { y_value = 0; break; }
217             case HBO::sweeping_location::FIFTY:
218             {
219                 double temp_y = 0;
220                 for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
221                 {
222                     if ( current_design.obtain_space(i).y + current_design.obtain_space(i).depth
                        > temp_y )
223                     {
224                         temp_y = current_design.obtain_space(i).y +
```

```
                        current_design.obtain_space(i).depth;
225                 }
226             }
227             y_value = temp_y * 0.5 ;
228             break;
229         }
230         case HBO::sweeping_location::HUNDRED:
231         {
232             double temp_y = 0 ;
233             for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
234             {
235                 if ( current_design.obtain_space(i).y + current_design.obtain_space(i).depth
                    > temp_y )
236                 {
237                     temp_y = current_design.obtain_space(i).y +
                        current_design.obtain_space(i).depth ;
238                 }
239             }
240             y_value = temp_y ;
241             break;
242         }
243         case HBO::sweeping_location::ARG_COUNT:
244         {
245             std::cout << "Error ARG_COUNT, exiting now... (Sweepings.hpp)" <<std::endl ;
                exit(1);
246         }
247         default:
248         {
249             std::cout << "Error default, exiting now... (Sweeping.hpp)" << std::endl ;
                exit(1) ;
250         }
251     }
252     // Check whether there are spaces at that y value
253     bool space_at_y = false ;
254     for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
255     {
256         if ( new_design.obtain_space(i).y <= y_value && new_design.obtain_space(i).y +
            new_design.obtain_space(i).depth > y_value )
257         {
258             space_at_y = true ;
259             break ;
260         }
261     }
262
263     int range = 10 ;
264     // If there is no space found at the x value, shift the value with small increments
        until a value is found for which a space is found
265     while ( !space_at_y )
266     {
267         if ( new_design.obtain_space_count() == 0 )
268         {
269             settings.terminate simulation = true ;
270             return new design ;
271         }
272         for ( unsigned int i = 0 ; i < new design.obtain space count() ; i++ )
273         {
274             if ( new design.obtain space(i).y <= ( y value - range ) &&
                new design.obtain space(i).y + new design.obtain space(i).depth > ( y value -
                range ) )
275             {
276                 space_at_y = true ;
277                 y_value -= range ;
278                 break ;
279             }
280             else if ( new_design.obtain_space(i).y <= ( y_value + range ) &&
                new_design.obtain_space(i).y + new_design.obtain_space(i).depth > ( y_value +
                range ) )
281             {
282                 space_at_y = true ;
283                 y_value += range ;
284                 break ;
285             }
286         }
287         range += 10 ;
288     } // end while() , note that no stopping criteria is formulated.
289     for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
290     {
291         if ( new_design.obtain_space(i).y <= y_value && new_design.obtain_space(i).y +
            new_design.obtain_space(i).depth > y_value )
292         {
293             area_swept += new_design.obtain_space(i).width *
                new_design.obtain_space(i).height ;
294         }
295     }
```

```
296         sweeping_distance = ( build_perform.get_initial_volume() - new_design.get_volume() ) /
            area_swept ;
297
298         new_design.sweep_y( y_value, sweeping_distance );
299
300         return new_design;
301     } // sweep_y_axis()
302
303     BSO::Spatial_Design::MS_Building sweep_z_axis( BSO::Spatial_Design::MS_Building&
            current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
            BSO::HBO::Settings& settings )
304     {
305         /*
306             Sweeps a building over the z axis
307         */
308         BSO::Spatial_Design::MS_Building new_design = current_design ;
309
310         double area_swept = 0 ;
311         double z_value ;
312
313         switch( settings.sweep_location )
314         {
315             case BSO::HBO::sweeping_location::ZERO: { z_value = 0 ; break ; }
316             case BSO::HBO::sweeping_location::FIFTY:
317             {
318                 double temp_z = 0 ;
319                 for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
320                 {
321                     if ( current_design.obtain_space(i).z +
                        current_design.obtain_space(i).height > temp_z )
322                     {
323                         temp_z = current_design.obtain_space(i).z +
                            current_design.obtain_space(i).height ;
324                     }
325                 }
326                 z_value = temp_z * 0.5 ;
327                 break ;
328             }
329             case BSO::HBO::sweeping_location::HUNDRED:
330             {
331                 double temp_z = 0 ;
332                 for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
333                 {
334                     if ( current_design.obtain_space(i).z +
                        current_design.obtain_space(i).height > temp_z )
335                     {
336                         temp_z = current_design.obtain_space(i).z +
                            current_design.obtain_space(i).height ;
337                     }
338                 }
339                 z value = temp z ;
340                 break ;
341             }
342             case BSO::HBO::sweeping location::ARG COUNT:
343             {
344                 std::cout << "Error ARG COUNT, exiting now... ( Sweeping.hpp)" << std::endl;
                    exit(1) ;
345             }
346             default:
347             {
348                 std::cout << "Error default, exiting now... ( Sweeping.hpp)" << std::endl ;
                    exit(1) ;
349             }
350         }
351
352         // Check whether there are spaces at that z value
353         bool space_at_z = false ;
354         for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
355         {
356             if ( new_design.obtain_space(i).z <= z_value && new_design.obtain_space(i).z +
                new_design.obtain_space(i).height > z_value )
357             {
358                 space_at_z = true ;
359                 break ;
360             }
361         }
362
363         int range = 10 ;
364
365         // If there is no space found at the z value, shift the value with small increments
            until a value is found for which a space is found
366         while ( !space_at_z )
367         {
368             if ( new_design.obtain_space_count() == 0 )
```

```
369             {
370                 settings.terminate_simulation = true ;
371                 return new_design ;
372             }
373             for ( unsigned int i = 0 ; i < new_design.obtain_space_count() ; i++ )
374             {
375                 if ( new_design.obtain_space(i).z <= ( z_value - range ) &&
                    new_design.obtain_space(i).z + new_design.obtain_space(i).height > ( z_value -
                    range ) )
376                 {
377                     space_at_z = true ;
378                     z_value -= range ;
379                     break ;
380                 }
381                 else if ( new_design.obtain_space(i).z <= ( z_value + range ) &&
                    new_design.obtain_space(i).z + new_design.obtain_space(i).height > ( z_value +
                    range ) )
382                 {
383                     space_at_z = true ;
384                     z_value += range ;
385                     break ;
386                 }
387             }
388             range += 10 ;
389         } // end while() , note that no stopping criteria is formulated.
390
391
392         for ( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
393         {
394             if ( current_design.obtain_space(i).z <= z_value && current_design.obtain_space(i).z
            + current_design.obtain_space(i).height > z_value )
395             {
396                 area_swept += current_design.obtain_space(i).width *
                    current_design.obtain_space(i).depth ;
397             }
398         }
399
400         double sweeping_distance = ( build_perform.get_initial_volume() -
            new_design.get_volume() ) / area_swept ;
401
402         new_design.sweep_z( z_value, sweeping_distance );
403
404         return new_design;
405     } // sweep_z_axis()
406
407 } // namespace Building_Modification
408 } // namespace HBO
409 } // namespace BSO
410
411 #endif //SWEEPING_HPP
```

```cpp
#ifndef SPLITTING_HPP
#define SPLITTING_HPP

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/HBO/Performance_Evaluation/Building_Performances.hpp>
#include <BSO/HBO/HBO_Settings.hpp>

#include <vector>
#include <cmath>

namespace BSO{ namespace HBO { namespace Building_Modification {

    BSO::Spatial_Design::MS_Building split_largest_space( BSO::Spatial_Design::MS_Building&,
    BSO::HBO::Performance_Evaluation::Building_Performances&, BSO::HBO::Settings& ) ;
    BSO::Spatial_Design::MS_Building split_smallest_space( BSO::Spatial_Design::MS_Building&,
    BSO::HBO::Performance_Evaluation::Building_Performances&, BSO::HBO::Settings& ) ;
    BSO::Spatial_Design::MS_Building split_best_space( BSO::Spatial_Design::MS_Building&,
    BSO::HBO::Performance_Evaluation::Building_Performances&, BSO::HBO::Settings& ) ;
    BSO::Spatial_Design::MS_Building split_worst_space( BSO::Spatial_Design::MS_Building&,
    BSO::HBO::Performance_Evaluation::Building_Performances&, BSO::HBO::Settings& ) ;
    void split_space_new( BSO::Spatial_Design::MS_Building&, int, Settings& ) ;


    BSO::Spatial_Design::MS_Building  split_spaces( BSO::Spatial_Design::MS_Building&
    current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    BSO::HBO::Settings& settings)
    {
        BSO::Spatial_Design::MS_Building new_design ;

        switch(settings.split)
        {
            case BSO::HBO::splitting_options::LARGEST: { new_design =
            split_largest_space(current_design, build_perform, settings ) ; break ; }
            // case BSO::HBO::splitting_options::SMALLEST: { new_design =
            split_smallest_space(current_design, build_perform, settings ) ; break ;}
            case BSO::HBO::splitting_options::BEST: { new_design =
            split_best_space(current_design, build_perform, settings ) ; break ; }
            case BSO::HBO::splitting_options::WORST: { new_design =
            split_worst_space(current_design, build_perform, settings ) ; break ; }
            case BSO::HBO::splitting_options::ARG_COUNT: { std::cout << "Error ARG_COUNT,
            exiting now... (Splitting.hpp)" << std::endl ; exit(1) ; }
            default: { std::cout << "Error default, exiting now... (Splitting.hpp)" <<
            std::endl; exit(1) ;}
        }

        return new_design;
    }

    BSO::Spatial_Design::MS_Building split_largest_space( BSO::Spatial_Design::MS_Building&
    current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    BSO::HBO::Settings& settings )
    {
        // function to find the space with the largest volume and split it. it prefers spaces
        with lower x coordinates
        BSO::Spatial Design::MS Building new design = current design;
        int spaces to split = ( build perform.get initial space count() -
        current design.obtain space count() );

        for ( unsigned int i = 0 ; i < spaces_to_split ; i++ ) // for the amount of spaces to be
        split
        {
            double largest_volume = 0 ;
            int space_index ;

            for ( unsigned int j = 0 ; j < new_design.obtain_space_count() ; j++ ) // find the
            largest volume
            {
                if ( new_design.obtain_space(j).get_volume() > largest_volume )
                {
                    largest_volume = new_design.obtain_space(j).get_volume() ;
                    space_index = j ;
                }
            }

            // if multiple spaces have the largest volume split the space with the lowest x
            coordinate
            std::vector<int> equal_spaces;

            equal_spaces.push_back(space_index);

            for ( unsigned int j = 0 ; j < new_design.obtain_space_count() ; j++ )
            {
                if ( new_design.obtain_space(j).get_volume() == largest_volume )
                {
```

```
                    equal_spaces.push_back(j);
                }
            }

            double lowest_z = new_design.obtain_space(equal_spaces[0]).z ;

            for ( unsigned int j = 0 ; j < equal_spaces.size() ; j++ )
            {
                if ( new_design.obtain_space(equal_spaces[j]).z < lowest_z )
                {
                    lowest_z = new_design.obtain_space(equal_spaces[j]).z ;
                    space_index = equal_spaces[j] ;
                }
            }

            // split the largest space
            split_space_new( new_design, space_index, settings ) ;
        }

        return new_design;
    } // split_largest_space

    BSO::Spatial_Design::MS_Building split_smallest_space( BSO::Spatial_Design::MS_Building&
    current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
    BSO::HBO::Settings& settings )
    {
        // function to find the smallest space and split it, prefers spaces with lower x
        coordinates
        BSO::Spatial_Design::MS_Building new_design = current_design ;
        int spaces_to_split = build_perform.get_initial_space_count() -
        current_design.obtain_space_count() ;

        for( unsigned int i = 0 ; i < spaces_to_split ; i++ ) // for the amount of spaces to split
        {
            double smallest_volume = 0 ;
            int space_index = 0;

            for ( unsigned int j = 0 ; j < new_design.obtain_space_count() ; j++ ) // find the
            smallest volume
            {
                if ( smallest_volume == 0 && ( new_design.obtain_space(j).width > 2000 ||
                new_design.obtain_space(j).depth > 2000 ) )
                {
                    smallest_volume = new_design.obtain_space(j).get_volume() ;
                    space_index = j ;
                }

                if( new_design.obtain_space(j).get_volume() < smallest_volume )
                {
                    if ( new_design.obtain_space( j ).width > 2000 || new_design.obtain_space( j
                    ).depth > 2000 )
                    {
                        smallest volume = new design.obtain space(j).get volume() ;
                        space index = j ;
                    }
                }
            }

            // if multiple spaces have the smallest volume split the space with the lowest x
            coordinate
            std::vector<int> equal_spaces;

            equal_spaces.push_back(space_index);

            for ( unsigned int j = 0 ; j < new_design.obtain_space_count() ; j++ )
            {
                if ( new_design.obtain_space(j).get_volume() == smallest_volume )
                {
                    equal_spaces.push_back(j);
                }
            }

            double lowest_x = new_design.obtain_space(equal_spaces[0]).x ;

            for ( unsigned int j = 0 ; j < equal_spaces.size() ; j++ )
            {
                if ( new_design.obtain_space(equal_spaces[j]).x < lowest_x )
                {
                    lowest_x = new_design.obtain_space(equal_spaces[j]).x ;
                    space_index = equal_spaces[j] ;
                }
            }

            // split the smallest space
```

```
142             split_space_new( new_design, space_index, settings ) ;
143             //new_design.split_space(space_index) ;
144         }
145
146         return new_design;
147     } // split_smallest_space
148
149     BSO::Spatial_Design::MS_Building split_best_space( BSO::Spatial_Design::MS_Building&
        current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
        BSO::HBO::Settings& settings )
150     {
151         BSO::Spatial_Design::MS_Building new_design = current_design ;
152         int spaces_to_split = build_perform.get_initial_space_count() -
            current_design.obtain_space_count();
153         BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform(
            current_design, build_perform ) ;
154
155         std::vector<int> selected_spaces;
156
157         for ( unsigned int i = 0 ; i < build_perform.groups.size() ; i++ )
158         {
159             if ( temp_build_perform.groups.empty() && selected_spaces.size() != spaces_to_split )
160             {
161                 std::cout<< "No more spaces to split while still more are required, exiting
                    now... " << std::endl ;
162                 settings.terminate_simulation = true ;
163                 return current_design ;
164             }
165
166             int best_group = temp_build_perform.best_group_modified() ;
167             for ( unsigned int j = 0 ; j < temp_build_perform.groups[ best_group
                ].space_ID.size() ; j++ )
168             {
169                 int space_index = current_design.get_space_index( temp_build_perform.groups[
                    best_group ].space_ID[ j ] ) ;
170                 if ( current_design.obtain_space( space_index ).width > 2000 ||
                    current_design.obtain_space( space_index ).depth > 2000 )
171                 {
172                     selected_spaces.push_back(
                        temp_build_perform.groups[temp_build_perform.best_group_modified()].space_ID[j
                        ] ) ;
173                 }
174
175                 if ( selected_spaces.size() == spaces_to_split )
176                 {
177                     break; // if enough spaces are selected for split end the loop
178                 }
179             }
180             if ( selected_spaces.size() == spaces_to_split )
181             {
182                 break ;
183             }
184
185             // if all space IDs of the best group are added remove the group so it can not be
                found again
186             temp build perform.delete group( temp build perform.best group modified() ) ;
187         }
188
189         for ( unsigned int i = 0 ; i < selected_spaces.size() ; i++ )
190         {
191             split_space_new( new_design, new_design.get_space_index( selected_spaces[ i ] ),
                settings ) ;
192             //new_design.split_space( new_design.get_space_index( selected_spaces[i] ) );
193         }
194
195         return new_design;
196     } // split_best_space
197
198     BSO::Spatial_Design::MS_Building split_worst_space( BSO::Spatial_Design::MS_Building&
        current_design, BSO::HBO::Performance_Evaluation::Building_Performances& build_perform,
        BSO::HBO::Settings& settings )
199     {
200         BSO::Spatial_Design::MS_Building new_design = current_design ;
201         int spaces_to_split = build_perform.get_initial_space_count() -
            current_design.obtain_space_count() ;
202         BSO::HBO::Performance_Evaluation::Building_Performances temp_build_perform(
            current_design, build_perform ) ;
203         std::vector<int> selected_spaces ;
204
205         for( unsigned int i = 0 ; i < build_perform.groups.size() ; i++ )
206         {
207             if ( temp_build_perform.groups.empty() && selected_spaces.size() != spaces_to_split )
208             {
209                 std::cout<< "No more spaces to split while still more are required, exiting
```

```
                now... " << std::endl ;
210             settings.terminate_simulation = true ;
211             return current_design ;
212         }
213
214         int worst_group_index = temp_build_perform.worst_group_modified() ;
215         //std::cout<< "Groups size: "<< temp_build_perform.groups[ worst_group_index
            ].space_ID.size() << std::endl;
216
217         for ( unsigned int j = 0 ; j < temp_build_perform.groups[ worst_group_index
            ].space_ID.size() ; j++ )
218         {
219
220             int space_index = new_design.get_space_index( temp_build_perform.groups[
                worst_group_index ].space_ID[ j ] ) ;
221
222             bool split_ok = false ;
223             if ( new_design.obtain_space( space_index ).width > 2000 ||
                new_design.obtain_space( space_index ).depth > 2000 )
224             {
225                 split_ok = true ;
226             }
227
228             if ( split_ok )
229                 selected_spaces.push_back( temp_build_perform.groups[ worst_group_index
                    ].space_ID[j] ) ;
230
231
232             if ( selected_spaces.size() == spaces_to_split )
233             {
234                 goto endloop;
235             }
236         }
237         // if all space_IDs of the worst group are added remove the group so it can not be
            found again
238         temp_build_perform.delete_group( worst_group_index ) ;
239     }
240     endloop:
241
242     for ( unsigned int i = 0 ; i < selected_spaces.size() ; i++ )
243     {
244         split_space_new( new_design, new_design.get_space_index( selected_spaces[ i ] ),
            settings ) ;
245         //new_design.split_space( new_design.get_space_index( selected_spaces[i] ) ) ;
246     }
247
248     return new_design ;
249 } // split_worst_space
250
251 void split_space_new( BSO::Spatial_Design::MS_Building& current_design, int space_index,
    Settings& settings )
252 {
253     switch ( settings.split pref )
254     {
255         case BSO::HBO::splitting preference::XYZ:
256         {
257             if ( current design.obtain space( space index ).width >= 0.9999 *
                current design.obtain space( space index ).depth && current design.obtain space(
                space_index ).width >= 0.9999 * current_design.obtain_space( space_index
                ).height )
258             {
259                 current_design.split_space_n( space_index, 0, 2 ) ;
260             }
261             else if ( current_design.obtain_space( space_index ).depth >=
                current_design.obtain_space( space_index).height  )
262             {
263                 current_design.split_space_n( space_index, 1, 2 ) ;
264             }
265             else
266             {
267                 current_design.split_space_n( space_index, 2, 2 ) ;
268             }
269             return ;
270         }
271         case BSO::HBO::splitting_preference::XZY:
272         {
273             if ( current_design.obtain_space( space_index ).width >= 0.9999 *
                current_design.obtain_space( space_index ).depth && current_design.obtain_space(
                space_index ).width >= 0.9999 * current_design.obtain_space( space_index
                ).height )
274             {
275                 current_design.split_space_n( space_index, 0, 2 ) ;
276             }
277             else if ( current_design.obtain_space( space_index ).height >= 0.9999 *
```

```
                    current_design.obtain_space( space_index ).depth )
278                 {
279                     current_design.split_space_n( space_index, 2, 2 ) ;
280                 }
281                 else
282                 {
283                     current_design.split_space_n( space_index, 1, 2 ) ;
284                 }
285                 return ;
286             }
287             case BSO::HBO::splitting_preference::YZX:
288             {
289                 if ( current_design.obtain_space( space_index ).depth >= 0.9999 *
                    current_design.obtain_space( space_index ).width && current_design.obtain_space(
                    space_index ).depth >= 0.9999 * current_design.obtain_space( space_index
                    ).height )
290                 {
291                     current_design.split_space_n( space_index, 1, 2 ) ;
292                 }
293                 else if ( current_design.obtain_space( space_index ).height >= 0.9999 *
                    current_design.obtain_space( space_index ).width )
294                 {
295                     current_design.split_space_n( space_index, 2, 2 ) ;
296                 }
297                 else
298                 {
299                     current_design.split_space_n( space_index, 0, 2 ) ;
300                 }
301                 return ;
302             }
303             case BSO::HBO::splitting_preference::YXZ:
304             {
305                 if ( current_design.obtain_space( space_index ).depth >= 0.9999 *
                    current_design.obtain_space( space_index ).width && current_design.obtain_space(
                    space_index ).depth >= 0.9999 * current_design.obtain_space( space_index
                    ).height )
306                 {
307                     current_design.split_space_n( space_index, 1, 2 ) ;
308                 }
309                 else if ( current_design.obtain_space( space_index ).width >= 0.9999 *
                    current_design.obtain_space( space_index ).height )
310                 {
311                     current_design.split_space_n( space_index, 0, 2 ) ;
312                 }
313                 else
314                 {
315                     current_design.split_space_n( space_index, 2, 2 ) ;
316                 }
317                 return ;
318             }
319             case BSO::HBO::splitting preference::ZYX:
320             {
321                 if ( current design.obtain space( space index ).height >= 0.9999 *
                    current design.obtain space( space index ).width && current design.obtain space(
                    space index ).height >= 0.9999 * current design.obtain space( space index
                    ).depth )
322                 {
323                     current design.split space n( space index, 2, 2 ) ;
324                 }
325                 else if ( current_design.obtain_space( space_index ).depth >= 0.9999 *
                    current_design.obtain_space( space_index ).width )
326                 {
327                     current_design.split_space_n( space_index, 1, 2 ) ;
328                 }
329                 else
330                 {
331                     current_design.split_space_n( space_index, 0, 2 ) ;
332                 }
333                 return ;
334             }
335             case BSO::HBO::splitting_preference::ZXY:
336             {
337                 if ( current_design.obtain_space( space_index ).height >= 0.9999 *
                    current_design.obtain_space( space_index ).width && current_design.obtain_space(
                    space_index ).height >= 0.9999 * current_design.obtain_space( space_index
                    ).depth )
338                 {
339                     current_design.split_space_n( space_index, 2, 2 ) ;
340                 }
341                 else if ( current_design.obtain_space( space_index ).width >= 0.9999 *
                    current_design.obtain_space( space_index ).depth )
342                 {
343                     current_design.split_space_n( space_index, 0, 2 ) ;
344                 }
```

```
345                 else
346                 {
347                     current_design.split_space_n( space_index, 1, 2 ) ;
348                 }
349                 return ;
350             }
351             case BSO::HBO::splitting_preference::ARG_COUNT:
352             {
353                 std::cout << "Error ARG_COUNT reached in split_space_new, exiting now... (
                    Splitting.hpp ) " << std::endl ;
354                 exit( 1 ) ;
355                 break ;
356             }
357             default:
358             {
359                 std::cout << "Error default reached in split_space_new, exiting now... (
                    Splitting.hpp ) " << std::endl ;
360                 exit( 1 ) ;
361             }
362         }
363     }
364
365 } // Building_Modification
366 } // HBO
367 } // BSO
368
369 #endif
```

Support Functions

```cpp
#ifndef NON_FEASIBLE_DESIGNS
#define NON_FEASIBLE_DESIGNS

#include <BSO/Spatial_Design/Movable_Sizable.hpp>
#include <BSO/Spatial_Design/Supercube.hpp>
#include <BSO/Spatial_Design/Conformation.hpp>
#include <BSO/Spatial_Design/Geometry/Rectangle.hpp>
#include <AEI Grammar/Grammar_2.hpp>
#include <BSO/HBO/HBO_Settings.hpp>

#include <vector>
#include <cmath>

namespace BSO { namespace HBO {

    /* Detect whether floating spaces occur within the given MS building design. Floating spaces are defined as a space which is not connected to the ground through shared surfaces
       two spaces connected thourgh a single edge are considered not interconnected*/
    bool detect_floating_space(BSO::Spatial_Design::MS_Building& current_design )
    {
        BSO::Spatial_Design::MS_Conformal CF(current_design, &(BSO::Grammar::grammar_2));

        // create a vector with lists of space IDs, the spaces_with_neighbours[i][0] id is the space from which perspective is viewed, the following numbers are the spaces connected to the space at 0
        std::vector< std::vector<int> > spaces_with_neighbours( current_design.obtain_space_count() ) ;

        // for all spaces in the conformal model
        for ( unsigned int i = 0 ; i < CF.get_space_count() ; i++ )
        {
            // get the space ID for which the connected spaces will be found
            spaces_with_neighbours[i].push_back( CF.get_space(i)->get_ID() ) ;

            // for all surfaces of a space ( 6 )
            for ( unsigned int j = 0 ; j < 6 ; j++ )
            {
                // for all rectangles that construct that surface
                for ( unsigned int k = 0 ; k < CF.get_space(i)->get_surface_ptr(j)->get_rectangle_count() ; k++ )
                {
                    BSO::Spatial_Design::Geometry::Rectangle* rectangle = CF.get_space(i)->get_surface_ptr(j)->get_rectangle_ptr(k) ;

                    // get all surfaces that are constructed with the rectangle
                    for ( unsigned int l = 0 ; l < rectangle->get_surface_count() ; l++ )
                    {
                        BSO::Spatial_Design::Geometry::Surface* surface = rectangle->get_surface_ptr(l) ;

                        if ( rectangle->get surface count() == 1 || rectangle->get surface count() == 2 )
                        {
                            if ( surface->get space ptr(0)->get ID() == CF.get space(i)->get ID() ) { /* if the surface belongs to the same space do nothing */ }
                            else
                            {
                                spaces_with_neighbours[i].push_back( surface->get_space_ptr(0)->get_ID() ) ;
                            }
                        }
                    }
                }
            }
        }

        std::vector<int> connected_spaces;

        // find all spaces that are connected to the ground ( z = 0 )

        for( unsigned int i = 0 ; i < current_design.obtain_space_count() ; i++ )
        {
            if( current_design.obtain_space(i).z == 0 )
            {
                connected_spaces.push_back(current_design.obtain_space(i).ID) ;

                for ( unsigned int j = 0 ; j < spaces_with_neighbours.size() ; j++ )
                {
                    if ( spaces_with_neighbours[j][0] == current_design.obtain_space(i).ID )
                    {
                        spaces_with_neighbours.erase( spaces_with_neighbours.begin() + j ) ;
                        break;
                    }
```

```
74                  }
75              }
76          }
77
78          while( !spaces_with_neighbours.empty() ) // as long as there are non connected spaces
79          {
80              beginning:
81              bool space_moved = false; // check whether a space moved to the connected group
82
83              for( unsigned int i = 0 ; i < spaces_with_neighbours.size() ; i++ )
84              {
85
86
87                  for ( unsigned int j = 1 ; j < spaces_with_neighbours[i].size() ; j++ )
88                  {
89
90                      for ( unsigned int k = 0 ; k < connected_spaces.size() ; k++ )
91                      {
92                          if ( spaces_with_neighbours[i][j] == connected_spaces[k] )
93                          {
94                              connected_spaces.push_back(spaces_with_neighbours[i][0]);
95
96                              spaces_with_neighbours.erase(spaces_with_neighbours.begin() + i) ;
97                              space_moved = true;
98                              i = 0; // atm gaat i naar -1 en daardoor word de for loop van j
                                gebroken
99                              j = 1;
100                             goto beginning;
101                         }
102                     }
103                 }
104             }
105
106             if ( space_moved == false && !spaces_with_neighbours.empty() ) // if in an entire
                    iteration none of the remaining spaces are moved from not connected to connected the
                    remaining spaces are floating
107             {
108                 return true;
109             }
110         }
111
112         if ( connected_spaces.size() == current_design.obtain_space_count() )
113         {
114             return false;
115         }
116         else
117         {
118             std::cout << "Error with floating spaces, exiting now...(Floating_Spaces.hpp)" <<
                    std::endl; exit(1);
119         }
120     } // detect floating space()
121
122     // In very rare cases the dimension of a space might go to infinity due to division by 0. It
        is not neccesary to check, however this function can be used when unknown errors occur
123     bool detect infinity( BSO::Spatial Design::MS Building& current design )
124     {
125         for ( unsigned int i = 0 ; i < current design.obtain space count() ; i++ )
126         {
127             BSO::Spatial_Design::MS_Space temp_space = current_design.obtain_space(i) ;
128             if( !std::isfinite( temp_space.x ) )
129                 return true ;
130             else if ( !std::isfinite( temp_space.y) )
131                 return true ;
132             else if ( !std::isfinite( temp_space.z) )
133                 return true ;
134             else if ( !std::isfinite( temp_space.width ) )
135                 return true ;
136             else if ( !std::isfinite( temp_space.depth ) )
137                 return true ;
138             else if ( !std::isfinite( temp_space.height ) )
139                 return true ;
140         }
141         return false ;
142     } // detect_infinity()
143
144     /* This function checks whether spaces of a building design are realistic, and if not how
        many unrealistic spaces a building contains
145        the level of unrealisticness is set in the settings file belonging to the simulation*/
146     void space_boundary_conditions_check( BSO::Spatial_Design::MS_Building& current_design,
        BSO::HBO::Settings& settings )
147     {
148
149         int unrealistic_spaces = 0 ;
150         for ( unsigned i = 0 ; i < current_design.obtain_space_count() ; i++ )
```

```
151          {
152
153              // if a space complies to any of these rules it is considered to be a space of
                 unrealistic dimensions
154              if( current_design.obtain_space(i).height > ( current_design.obtain_space(i).width +
                 current_design.obtain_space(i).depth ) / 2.0 &&
                 current_design.obtain_space(i).height > 3000 )
155                  unrealistic_spaces++ ;
156              else if ( current_design.obtain_space(i).depth > 20 *
                 current_design.obtain_space(i).width )
157                  unrealistic_spaces++ ;
158              else if ( current_design.obtain_space(i).width > 20 *
                 current_design.obtain_space(i).depth )
159                  unrealistic_spaces++ ;
160
161          }
162          settings.realistic_building = ((float)current_design.obtain_space_count() -
             (float)unrealistic_spaces )/ (float)current_design.obtain_space_count() ;
163
164          return ;
165      }
166
167  } // namespace HBO
168  } // namespace BSO
169
170  #endif // NON_FEASIBLE_DESIGNS
```

Movable-Sizable Building

```cpp
#ifndef MOVABLE_SIZABLE_HPP
#define MOVABLE_SIZABLE_HPP

#include <BSO/Spatial_Design/Supercube.hpp> // for operator overloading, this also loads in <string> and <vector>
#include <BSO/Trim_And_Cast.hpp>

#include <boost/tokenizer.hpp>
#include <boost/algorithm/string.hpp>

#include <iostream>
#include <fstream>
#include <stdexcept>
#include <algorithm> // for vector::sort()
#include <cmath>
#include <cstdlib>

namespace BSO { namespace Spatial_Design
{

    /*
     *
     */

    // Structure definition:

    struct MS_Space
    {
        int ID; // space identification number
        std::string m_space_type;
        double width, depth, height; // room dimensions
        double x, y, z; // room coordinates (closest to origin)
        bool surfaces_given; // TG_22-03-2017  checks if surface types of ALL cardinal directions are given, false if incomplete
        bool space_type_given;
        std::string surface_type [6]; // TG_22-03-2017  cardinal directions {north, east, south, west, top, bottom}
        double get_area()
        {
            return width*depth;
        }
        double get_volume()
        {
            return width*depth*height;
        }
        double get_aspect_ratio()
        {
            return (width>depth) ? width/depth : depth/width;
        }

        void init zero()
        {
            ID = 0;
            width = 0;
            depth = 0;
            height = 0;
            x = 0.0;
            y = 0.0;
            z = 0.0;
            m_space_type = "";
            surfaces_given = false;
            space_type_given = false;

            for (int i = 0; i < 6; i++) // TG_22-03-2017  clear the temp_space.surface_types to obtain an empty in case of invalid amount of surface types
            {
                surface_type[i] = "";
            }
        }

        MS_Space()
        {
            init_zero();

        }

    }; // MS_Space



    /*
     *
     */

```

```
81      // Class definition:
82
83      class MS_Building
84      {
85      private:
86          std::vector<MS_Space> m_spaces;
87          int m_last_space_ID;
88      public:
89          MS_Building(SC_Building); // ctor, converts from Supercube to Movable Sizable
90          MS_Building(std::string); // ctor, initializes the class by reading an input file
91          MS_Building(); //ctor, creates an empty MS_Build object
92          ~MS_Building(); // dtor
93
94          void read_file(std::string); // reads an input file (e.g. "Filename.txt")
95          void write_file(std::string); // writes an output file (e.g. "Filename.txt")
96
97          operator SC_Building() const; // converts from Movable Sizable to Supercube
98
99          int obtain_space_count(); // returns the size of the rooms vector
100         double get_volume(); // returns the volume of the building
101
102         MS_Space obtain_space(int space_index); // returns a MS_Space structure from the
            m_spaces vector at a certain index
103         int get_space_index(int space_ID); // returns the index from the m_spaces vector if the
            space ID is found
104
105         int get_space_ID(unsigned int space_index); // NEW returns the space ID if the index
            from the m_spaces vector is found
106         std::string get_space_type(unsigned int space_index);
107         std::string get_surface_type(int space_index, int surface_type); // NEW returns the
            surface_type from the m_spaces vector at a certain index
108         bool get_surfaces_given(int space_index); // NEW returns boolean surfaces_given from the
            m_spaces vector at a certain index
109
110         void add_space(MS_Space); // adds a space to the building
111         void delete_space(int space_index); // deletes a space from the building
112
113         void split_space(int space_index); // splits a space across its largest dimensions
114         void split_space(int space_index, int axis); // splits in the middle across axis:
            [0,1,2] for respectively [x,y,z]
115         void split_space_n(int space_index, int axis, unsigned int n_divisions); // a space into
            n equal parts, across axis: [0,1,2] for respectively [x,y,z]
116
117         void scale(double n, int i); // scales the building design about the {i} axis;
118         void scale_x(double n); // scales the building design about the x axis
119         void scale_y(double n); // scales the building design about the y axis
120         void scale_z(double n); // scales the building design about the z axis
121
122         void snap_on(double m); // snaps the coordinates and dimensions of the spatial designs
            onto a grid defined by multiples of 'm'
123         void snap on(double m, int axis); // snaps the coordinates and dimensions of the spatial
            designs onto a grid defined by multiples of 'm'
124
125         void clear design();
126
127         void reset z zero(); // resets the z-coordinates in the building design if there are no
            z-coordinates equal to zero
128
129         bool check_cell(SC_Building S, int cell_index, int space_ID,
130                         double x_origin, double y_origin, double z_origin) const; // checks if a
                            cell of Supercube S belongs to a certain room_ID, also requires the x-
                            y- and z- coordinates of the supercube (first values of x_, y_ and
                            z_values vectors)
131
132         // TS added functions
133         int get_last_space_id(); // retrieves the last space ID used in the MS Building
134         void search_last_space_id(); // searches in the MS Building for the last used space id
135
136         void sweep_x( double location, double distance );
137         void sweep_y( double location, double distance );
138         void sweep_z( double location, double distance );
139
140     }; // MS_Building
141
142
143
144     // Implementation of the member functions:
145
146     MS_Building::MS_Building(SC_Building S) //conversion from Supercube to MovableSizable
147     {
148         m_last_space_ID = -1;
149         int w_origin = S.w_size(); // initialize the indexes which will contain the origin of
            the Movable Sizable representation
150         int d_origin = S.d_size(); // these are initialized to the cell containing the largest
```

```
        coordinates, to be updated later
151     int h_origin = S.h_size();
152
153     std::vector<double> w_coord_values(S.w_size()+1);
154     std::vector<double> d_coord_values(S.d_size()+1);
155     std::vector<double> h_coord_values(S.h_size()+1);
156
157     // find which cell willcontain the origin
158     for (unsigned int cell_index = 1; cell_index <= (S.w_size()*S.d_size()*S.h_size());
        cell_index++) // starts with 1, since first index is the room ID // each cell is checked
        for each room, whether it describes a room
159     { // for each cell
160         int w_index = S.get_w_index(cell_index);  // compute the grid indexes of the
            considered cell
161         int d_index = S.get_d_index(cell_index);
162         int h_index = S.get_h_index(cell_index);
163
164         for (unsigned int i = 0; i < S.b_size(); i++) // check each room for the considered
            cell, whether it describes a room
165         {
166             if (S.request_b(i, cell_index) == 1) // if it does describe a room, then update
                the origin indexes
167                 {
168                     if (w_index < w_origin) { w_origin = w_index; } // update the indexes
                        containing the origin of the MS representation
169                     if (d_index < d_origin) { d_origin = d_index; }
170                     if (h_index < h_origin) { h_origin = h_index; }
171                 }
172         }
173     }
174
175     // compute the global coordinates for each cell's origin
176     w_coord_values[w_origin] = 0;
177     for (unsigned int i = w_origin+1; i < w_coord_values.size(); i++)
178     {
179         w_coord_values[i] = w_coord_values[i-1] + floor(S.request_w(i-1));
180     }
181
182     d_coord_values[d_origin] = 0;
183     for (unsigned int i = d_origin+1; i < d_coord_values.size(); i++)
184     {
185         d_coord_values[i] = d_coord_values[i-1] + floor(S.request_d(i-1));
186     }
187
188     h_coord_values[h_origin] = 0;
189     for (unsigned int i = h_origin+1; i < h_coord_values.size(); i++)
190     {
191         h_coord_values[i] = h_coord_values[i-1] + floor(S.request_h(i-1));
192     }
193
194     // compute each MS space
195     for (unsigned int i = 0; i < S.b size(); i++)
196     {
197         MS Space temp space; // initializing all values in an object of the RoomMS structure
198         temp space.ID = i + 1; // assigning the room ID to the RoomMS object
199         int maximum = 0, minimum = 0;   // initializing minimum and maximum indexes of the
            space with id: i+1
200         for (unsigned int cell index = 0; cell index < S.b row size(i); cell index++)   //
            finds the min and max indexes
201         {
202             if (minimum == 0 && S.request_b(i, cell_index) == 1)    // finds first
                cell_index with value 1 in the cell_vector, this is the cell containing the
                room's origin assuming spaces are cuboid
203                 {minimum = cell_index;}
204             if (S.request_b(i, cell_index) == 1)    // finds the last index with value 1 in
                the cell_array, this is the room's outmost cell assuming spaces are cuboid
205                 {maximum = cell_index;}
206         }
207
208         int min_w = S.get_w_index(minimum), max_w = S.get_w_index(maximum),
209             min_d = S.get_d_index(minimum), max_d = S.get_d_index(maximum),
210             min_h = S.get_h_index(minimum), max_h = S.get_h_index(maximum);   // computes
                the grid indexes of the min and max cells
211
212         temp_space.x = w_coord_values[min_w];
213         temp_space.width = w_coord_values[max_w] + floor(S.request_w(max_w)) - temp_space.x;
214         temp_space.y = d_coord_values[min_d];
215         temp_space.depth = d_coord_values[max_d] + floor(S.request_d(max_d)) - temp_space.y;
216         temp_space.z = h_coord_values[min_h];
217         temp_space.height = h_coord_values[max_h] + floor(S.request_h(max_h)) - temp_space.z;
218
219         m_spaces.push_back(temp_space); // stack the RoomMS structure in the rooms vector
220         if (m_last_space_ID < m_spaces.back().ID)
221         {
```

```
222                 m_last_space_ID = m_spaces.back().ID;
223             }
224         }
225     } //ctor
226
227     MS_Building::MS_Building(std::string filename)
228     {
229         m_last_space_ID = -1;
230
231         if (filename.empty())
232             throw std::runtime_error("No file name specified to initialise a MS building spatial
                design");
233
234
235         read_file(filename);//ctor
236     } // ctor
237
238     MS_Building::MS_Building()
239     {
240
241     }
242
243
244     MS_Building::~MS_Building()
245     {
246
247     } // dtor
248
249     void MS_Building::read_file(std::string file_name)
250     {
251         std::ifstream input(file_name.c_str()); // initialize input stream from file: file_name
252         if (!input.is_open() && file_name != "empty")
253             throw std::runtime_error("Could not open file: " + file_name);
254
255         std::string line;
256         boost::char_separator<char> sep(","); // defines what separates tokens in a string
257         typedef boost::tokenizer< boost::char_separator<char> > t_tokenizer; // settings for the
            boost::tokenizer
258
259         while (!input.eof() && file_name != "empty") // continue while the End Of File has not
            been reached
260         {
261
262             getline(input,line); // get next line from the file
263             boost::algorithm::trim(line); // remove white space from start and end of line (to
                see if it is an empty line, remove any incidental white space)
264             if (line == "") //skip empty lines (tokenizer does not like it)
265             {
266                 continue; // continue to next line
267             }
268
269             t_tokenizer tok(line, sep); // tokenize the line
270             t_tokenizer::iterator token = tok.begin(); // set iterator to first token
271             int number_of_tokens = std::distance( tok.begin(), tok.end() ); // TG 22-03-2017
                determine number of tokens within a line of the input file
272
273             if (*token != "R")
274             {
275                 continue; // continue to next line in text file
276             }
277             else // if the first token is an "R" then this line describes a space
278             {
279                 MS_Space temp_space; // this MS_Space structure will temporarily hold the space
                    described by the considered line
280                 token++; // this is the 'ID'
281                 temp_space.ID = trim_and_cast_int(*token);
282
283                 token++; // this is 'width'
284                 temp_space.width = trim_and_cast_double(*token);
285
286                 token++; // this is 'depth'
287                 temp_space.depth = trim_and_cast_double(*token);
288
289                 token++; // this is 'height'
290                 temp_space.height = trim_and_cast_double(*token);
291
292                 token++; // this is 'x-coordinate'
293                 temp_space.x = trim_and_cast_double(*token);
294
295                 token++; // this is 'y-coordinate'
296                 temp_space.y = trim_and_cast_double(*token);
297
298                 token++; // this is 'z-coordinate'
299                 temp_space.z = trim_and_cast_double(*token);
```

```
300
301                 switch (number_of_tokens) // TG_22-03-2017  defines boolean of surface given
302                 {
303                 case 8:
304                 {
305                     break;
306                 }
307                 case 9:
308                 {
309                     token++; // space_type
310                     temp_space.m_space_type = *token;
311                     boost::algorithm::trim(temp_space.m_space_type);
312                     temp_space.space_type_given = true;
313                     break;
314                 }
315                 case 15:
316                 {
317                     token++; // space_type
318                     temp_space.m_space_type = *token;
319                     boost::algorithm::trim(temp_space.m_space_type);
320                     temp_space.space_type_given = true;
321                     // NOTE no break, so we continue to the next case!
322                 }
323                 case 14:
324                 {
325                     token++; // TG_22-03-2017  this is 'north-surface'
326                     temp_space.surface_type[0] = *token;
327                     boost::algorithm::trim(temp_space.surface_type[0]);
328
329                     token++; // TG_22-03-2017  this is 'east-surface'
330                     temp_space.surface_type[1] = *token;
331                     boost::algorithm::trim(temp_space.surface_type[1]);
332
333                     token++; // TG_22-03-2017  this is 'south-surface'
334                     temp_space.surface_type[2] = *token;
335                     boost::algorithm::trim(temp_space.surface_type[2]);
336
337                     token++; // TG_22-03-2017  this is 'west-surface'
338                     temp_space.surface_type[3] = *token;
339                     boost::algorithm::trim(temp_space.surface_type[3]);
340
341                     token++; // TG_22-03-2017  this is 'top-surface'
342                     temp_space.surface_type[4] = *token;
343                     boost::algorithm::trim(temp_space.surface_type[4]);
344
345                     token++; // TG_22-03-2017  this is 'bottom-surface'
346                     temp_space.surface_type[5] = *token;
347                     boost::algorithm::trim(temp_space.surface_type[5]);
348
349                     temp_space.surfaces_given  = true;
350                     break;
351                 }
352                 default:
353                 {
354                     std::cerr << "Space ID " << temp space.ID << " contains invalid amount of
                        tokens. (Movable Sizable.hpp)" << std::endl;
355                     exit(1);
356                     break;
357                 }
358                 };
359
360                 m_spaces.push_back(temp_space); // stack the MS_Space structure in the m_spaces
                    vector
361
362                 if (m_last_space_ID < m_spaces.back().ID)
363                 {
364                     m_last_space_ID = m_spaces.back().ID;
365                 }
366             }
367         }
368     } // read_file()
369
370     void MS_Building::write_file(std::string filename)
371     {
372         std::ofstream output; // initialize output stream
373         output.open(filename.c_str(), std::ofstream::out | std::ofstream::trunc); // stream to
            file: filename
374
375         for (unsigned int i = 0; i < m_spaces.size(); i++)  // for each room write the
            description as follows:
376
                                                                //R,width,depth,height,x-coor,y-coor,z
                                                                -coor,north-surf, east-surf,
                                                                south-surf, west-surf, top-surf,
```

```
                                                                bottom-surf
            {
            output << "R," << m_spaces[i].ID    << "," << m_spaces[i].width
                   << ","  << m_spaces[i].depth << "," << m_spaces[i].height
                   << ","  << m_spaces[i].x     << "," << m_spaces[i].y
                   << ","  << m_spaces[i].z;
            if (m_spaces[i].space_type_given == true)
            {
                output << "," << m_spaces[i].m_space_type;
            }
            if (m_spaces[i].surfaces_given == true) // TG_22-03-2017
            {
            output << "," << m_spaces[i].surface_type[0] << "," << m_spaces[i].surface_type[1]
                   << "," << m_spaces[i].surface_type[2] << "," << m_spaces[i].surface_type[3]
                   << "," << m_spaces[i].surface_type[4] << "," << m_spaces[i].surface_type[5];
            }
            output << std::endl;
            }

        output.close(); // terminate the output stream
    } // write_file()

    int MS_Building::obtain_space_count()
    {
        return m_spaces.size();
    } // obtain_space_count()

    double MS_Building::get_volume()
    {
        double volume = 0;
        for (unsigned int i = 0; i < m_spaces.size(); i++)
        {
            volume += m_spaces[i].get_volume();
        }
        return volume;
    } // get_volume()

    MS_Space MS_Building::obtain_space(int space_index)
    {
        return m_spaces[space_index];
    } // obtain_room()

    int MS_Building::get_space_index(int space_ID)
    {
        for (unsigned int i = 0; i < m_spaces.size(); i ++)
        {
            if (space_ID == m_spaces[i].ID)
            {
                return i;
            }
        }
        std::cerr << "Could not find space by its ID (Movable Sizable.hpp), exiting now... " <<
        std::endl;
        exit(1);
    }

    std::string MS Building::get space type(unsigned int space index)
    {
        return m_spaces[space_index].m_space_type;
    } // get_sapce_type()

    int MS_Building::get_space_ID(unsigned int space_index) // NEW
    {
        if (space_index >= m_spaces.size()) // NEW error if requested space index is not valid
            {
            std::cerr << "Could not find space ID by its index (Movable_Sizable.hpp), exiting
            now... " << std::endl;
            exit(1);
            }
        {
            return m_spaces[space_index].ID;
        }
    } // NEW get_space_ID()

    std::string MS_Building::get_surface_type(int space_index, int surface_type) // NEW
    {
        return m_spaces[space_index].surface_type[surface_type]; // cardinal directions
        {0=north, 1=east, 2=south, 3=west, 4=top, 5=bottom}
    } // NEW get_surface_type()

    bool MS_Building::get_surfaces_given(int space_index) // NEW
    {
        return m_spaces[space_index].surfaces_given; // returns true if surface types of ALL
        cardinal directions are given
```

```
456     } // get_surfaces_given()
457
458     void MS_Building::add_space(MS_Space space)
459     {
460         m_spaces.push_back(space);
461         if (m_last_space_ID < m_spaces.back().ID)
462         {
463             m_last_space_ID = m_spaces.back().ID;
464         }
465     }
466
467     void MS_Building::delete_space(int space_index)
468     {
469         m_spaces.erase(m_spaces.begin() + space_index);
470     }
471
472     void MS_Building::split_space(int space_index)
473     {
474         if ( m_spaces[space_index].height < 6000 )
475         {
476             if (m_spaces[space_index].width < 0.9999 * m_spaces[space_index].depth)
477             { // if the space is deeper than its width
478                 this->split_space_n(space_index, 1, 2) ;
479             }
480             else
481             { // if the space is wider than its depth or as wide as deep
482                 this->split_space_n(space_index, 0, 2) ;
483             }
484         }
485         else
486         {
487             if( m_spaces[space_index].width < 0.9999 * m_spaces[space_index].depth &&
                m_spaces[space_index].height < 0.9999 * m_spaces[space_index].depth )
488             { // if the space is deeper than its width and height
489                 this->split_space_n(space_index, 1, 2 ) ;
490             }
491             else if ( m_spaces[space_index].depth < 0.9999 * m_spaces[space_index].width &&
                m_spaces[space_index].height < 0.9999 * m_spaces[space_index].width )
492             {
493                 this->split_space_n(space_index, 0, 2 ) ;
494             }
495             else
496             {
497                 this->split_space_n(space_index, 2, 2 ) ;
498             }
499         }
500     } // split_space()
501
502     void MS_Building::split_space(int space_index, int axis)
503     {
504         this->split space n(space index, axis, 2);
505     } // split space()
506
507     void MS Building::split space n(int space index, int axis, unsigned int n division)
508     {
509         if (n division < 2)
510         {
511             return ;
512         }
513         MS_Space temp = m_spaces[space_index];
514         double * loc = nullptr ;
515         double * dim = nullptr ;
516
517         if (axis == 0)
518         {
519             loc = &temp.x ;
520             dim = &temp.width ;
521         }
522         else if (axis == 1)
523         {
524             loc = &temp.y;
525             dim = &temp.depth;
526         }
527         else if (axis == 2)
528         {
529             loc = &temp.z;
530             dim = &temp.height;
531         }
532         else
533         {
534             throw std::invalid_argument("In function split_space_n (Movable_Sizable.hpp):
                expected the axis number to be either 0, 1, or 2.");
535         }
536
```

```
537             double delta = floor(*dim / (double)n_division);
538             double init_loc = *loc;
539             double init_dim = *dim;
540 
541             for (unsigned int i = 0; i < n_division; i++)
542             {
543                 temp.ID = ++m_last_space_ID;
544                 *dim = delta;
545                 if (i == n_division - 1)
546                 {
547                     *dim = (init_dim + init_loc) - (*loc);
548                 }
549 
550                 m_spaces.push_back(temp);
551                 *loc = *loc + *dim;
552             }
553             delete_space(space_index);
554         } // split_space_n()
555 
556         void MS_Building::scale(double n, int axis)
557         {
558             for (unsigned int i = 0; i < m_spaces.size(); i++)
559             { // scale each space
560                 MS_Space * space_ptr = &m_spaces[i];
561                 double * loc = nullptr;
562                 double * dim = nullptr;
563 
564                 // scale space i in the selected axis
565                 if (axis == 0)
566                 {
567                     loc = &(space_ptr->x);
568                     dim = &(space_ptr->width);
569                 }
570                 else if (axis == 1)
571                 {
572                     loc = &(space_ptr->y);
573                     dim = &(space_ptr->depth);
574                 }
575                 else if (axis == 2)
576                 {
577                     loc = &(space_ptr->z);
578                     dim = &(space_ptr->height);
579                 }
580                 else
581                 {
582                 throw std::invalid_argument("In function split_space_n (Movable_Sizable.hpp):
                    expected the axis number to be either 0, 1, or 2.");
583                 }
584 
585                 // scale with the selected scaling factor n
586                 double temp sum  = *loc + *dim; // calculate the coordinate of the end of the space
                    in that axis
587                 temp sum = round(temp sum * n); // scale the end coordinate
588                 *loc = round(*loc * n); // scale the begin coordinate
589 
590                 *dim = temp sum - *loc; // calculate the dimension from the end and begin coordinate
591             }
592 
593         }
594 
595         void MS_Building::scale_x(double n)
596         {
597             scale(n,0);
598         }
599 
600         void MS_Building::scale_y(double n)
601         {
602             scale(n,1);
603         }
604 
605         void MS_Building::scale_z(double n)
606         {
607             scale(n,2);
608         }
609 
610         void MS_Building::snap_on(double m)
611         {
612             snap_on(m,0);
613             snap_on(m,1);
614             snap_on(m,2);
615         } // snap_on()
616 
617         void MS_Building::snap_on(double m, int axis)
618         {
```

```
619         for (unsigned int i = 0; i < m_spaces.size(); i++)
620         { // snap each space
621         MS_Space * space_ptr = &m_spaces[i];
622             double * loc = nullptr;
623             double * dim = nullptr;
624 
625             // snap dimensions and locations in the selected axis
626             if (axis == 0)
627             {
628                 loc = &(space_ptr->x);
629                 dim = &(space_ptr->width);
630             }
631             else if (axis == 1)
632             {
633                 loc = &(space_ptr->y);
634                 dim = &(space_ptr->depth);
635             }
636             else if (axis == 2)
637             {
638                 loc = &(space_ptr->z);
639                 dim = &(space_ptr->height);
640             }
641             else
642             {
643             throw std::invalid_argument("In function split_space_n (Movable_Sizable.hpp):
                expected the axis number to be either 0, 1, or 2.");
644             }
645 
646             double n_loc = *loc + *dim;
647 
648             *loc  = round((*loc) / m) * m;
649             n_loc = round(n_loc  / m) * m;
650             *dim  = n_loc - *loc;
651         }
652 
653     } // snap_on()
654 
655     MS_Building::operator SC_Building() const // conversion from MovableSizable to Supercube
656     {
657         SC_Building S; // initialize an object of class Supercube, information will be added and
            then it will be returned at the end
658         std::vector<double> x_values, y_values, z_values; // these vectors will contain all x-
            y- and z-values of the MS representation
659 
660         for (unsigned int i = 0; i < m_spaces.size(); i++) // this stacks every x- y- and
            z-coordinate of every room in the respective vectors
661         {
662             x_values.push_back(m_spaces[i].x);
663             x_values.push_back(m_spaces[i].x+m_spaces[i].width);
664             y_values.push_back(m_spaces[i].y);
665             y values.push back(m spaces[i].y+m spaces[i].depth);
666             z values.push back(m spaces[i].z);
667             z values.push back(m spaces[i].z+m spaces[i].height);
668         }
669 
670         sort(x values.begin(), x values.end()); // sorts all values in the vectors in ascending
            order
671         sort(y values.begin(), y values.end());
672         sort(z_values.begin(), z_values.end());
673         x_values.erase(unique(x_values.begin(), x_values.end()), x_values.end()); // erases all
            duplicates from the vectors
674         y_values.erase(unique(y_values.begin(), y_values.end()), y_values.end());
675         z_values.erase(unique(z_values.begin(), z_values.end()), z_values.end());
676 
677         double x_origin = x_values[0], y_origin = y_values[0], z_origin = z_values[0]; // saves
            the coordinates of the super cube's origin
678 
679         for (unsigned int i = 0; i < x_values.size()-1; i++)    // computes widths of super cube
            grid and puts them in the w_values vector
680             { S.stack_w_value(x_values[i+1] - x_values[i]); }
681         for (unsigned int i = 0; i < y_values.size()-1; i++)    // computes depths of super cube
            grid and puts them in the d_values vector
682             { S.stack_d_value(y_values[i+1] - y_values[i]); }
683         for (unsigned int i = 0; i < z_values.size()-1; i++)    // computes heights of super
            cube grid and puts them in the h_values vector
684             { S.stack_h_value(z_values[i+1] - z_values[i]); }
685 
686         int cube_size = S.w_size()*S.d_size()*S.h_size(); // initialize variable containing
            super cube size
687         std::vector<int> b_values_row; // initializes a vector for the b_values matrix
688 
689         for (unsigned int i = 0; i < m_spaces.size(); i++)  // computes a row of the b_values
            matrix and adds these to the matrix for each room
690         {
```

```
691             b_values_row.clear(); // clears the vector's data from previous iteration
692             b_values_row.push_back(i + 1); // index 0 contains the room ID, for the super cube
                the count starts again from 1.
693
694             for (int cell_index = 1; cell_index <= cube_size; cell_index++) // checks each cell
                within the super cube if it belongs to room with ID: i
695             {
696                 b_values_row.push_back((check_cell(S, cell_index, i, x_origin, y_origin,
                    z_origin) ? 1 : 0)); // If a cell belongs to the room assign 1 to index, if not 0
697             }
698
699             S.stack_b_value_row(b_values_row); // adds the row to the b_values matrix
700         }
701
702         return S;
703     } // operator SC_Building() const
704
705     void MS_Building::clear_design()
706     {
707         m_spaces.clear();
708         m_last_space_ID = -1;
709     } // clear_design()
710
711     void MS_Building::reset_z_zero()
712     {
713         int min = m_spaces[0].z;
714         // find the minimum value of the z-coordinates in the building
715         for (unsigned int i = 1; i < m_spaces.size(); i++)
716         { // for each space
717             if (m_spaces[i].z < min)
718             { // check if this space's z is smaller than min
719                 min = m_spaces[i].z; // if yes, assign its value to min
720             }
721         }
722
723         for (unsigned int i = 0; i < m_spaces.size(); i++)
724         { // for each space
725             m_spaces[i].z -= min; // shift the z-coordinates down by min
726         }
727     } // reset_z_zero()
728
729     bool MS_Building::check_cell(SC_Building S, int cell_index, int space_index, double
        x_origin, double y_origin, double z_origin) const
730     {
731         int w_index = S.get_w_index(cell_index); // compute the indexes of cell with index:
            cell_index
732         int d_index = S.get_d_index(cell_index);
733         int h_index = S.get_h_index(cell_index);
734         double x_coor = x_origin, y_coor = y_origin, z_coor = z_origin; // initialize the
            coordinates of the cell's origin to the coordinates of the super cube's origin
735
736         for (int k = 0; k < w index; k++)    // update the x coordinates of the cell's origin
737             { x coor += S.request w(k); }
738         for (int k = 0; k < d index; k++)    // update the y coordinates of the cell's origin
739             { y coor += S.request d(k); }
740         for (int k = 0; k < h index; k++)    // update the z coordinates of the cell's origin
741             { z coor += S.request h(k); }
742
743         if((x_coor >= m_spaces[space_index].x && x_coor < m_spaces[space_index].x +
            m_spaces[space_index].width) &&
744            (y_coor >= m_spaces[space_index].y && y_coor < m_spaces[space_index].y +
               m_spaces[space_index].depth) &&
745            (z_coor >= m_spaces[space_index].z && z_coor < m_spaces[space_index].z +
               m_spaces[space_index].height))  // if a cell is within the bounds of space r return
               true
746             { return true; }
747         else
748             { return false; } // if not return false
749     } // check_cell()
750
751
752
753     int MS_Building::get_last_space_id()
754     {
755         return m_last_space_ID;
756     }
757
758     void MS_Building::search_last_space_id()
759     {
760         m_last_space_ID = 0;
761
762         for( unsigned int i = 0 ; i < m_spaces.size() ; i++)
763         {
764             if(m_spaces[i].ID > m_last_space_ID)
```

```
765                 {
766                     m_last_space_ID = m_spaces[i].ID;
767                 }
768             }
769             return;
770         }
771
772         void MS_Building::sweep_x( double location, double distance )
773         {
774             for( unsigned int i = 0 ; i < m_spaces.size() ; i++ )
775             {
776                 if( m_spaces[ i ].x + m_spaces[ i ].width < location ) { }
777                 else if ( m_spaces[i].x <= location && m_spaces[ i ].x + m_spaces[ i ].width >
                    location )
778                 {
779                     m_spaces[ i ].width += distance ;
780                 }
781                 else if ( m_spaces[ i ].x > location )
782                 {
783                     m_spaces[ i ].x += distance;
784                 }
785             }
786         }
787
788         void MS_Building::sweep_y( double location, double distance )
789         {
790             for( unsigned int i = 0 ; i < m_spaces.size() ; i++ )
791             {
792                 if( m_spaces[ i ].y + m_spaces[ i ].depth < location ) { }
793                 else if ( m_spaces[ i ].y <= location && m_spaces[ i ].y + m_spaces[ i ].depth >
                    location )
794                 {
795                     m_spaces[ i ].depth += distance ;
796                 }
797                 else if ( m_spaces[ i ].y > location )
798                 {
799                     m_spaces[ i ].y += distance;
800                 }
801             }
802         }
803
804         void MS_Building::sweep_z( double location, double distance )
805         {
806             for( unsigned int i = 0 ; i < m_spaces.size() ; i++ )
807             {
808                 if( m_spaces[ i ].z + m_spaces[ i ].height < location ) { }
809                 else if ( m_spaces[ i ].z <= location && m_spaces[ i ].z + m_spaces[ i ].height >
                    location )
810                 {
811                     m_spaces[ i ].height += distance ;
812                 }
813                 else if ( m spaces[ i ].z > location )
814                 {
815                     m spaces[ i ].z += distance;
816                 }
817             }
818         }
819
820
821     } // namespace Spatial_Design
822     } // namespace BSO
823     #endif // MOVABLE_SIZABLE_HPP
824
```

D Cube versus Sphere in orthogonal space

The building physics discipline as implemented in this research mainly operates on the relation between surface area and volume. The perfect shape in an ideal world for this ratio is spherical, however, the BSO toolbox operates in a design search space where only orthogonal shapes are possible. While no perfect spheres can exist in an orthogonal space, approximations can be constructed using a granulation of the surface area. Figure D.1 illustrates three circles with different a granulation, while they provide a representation of a circle it is not the optimal ratio of circumference versus area.

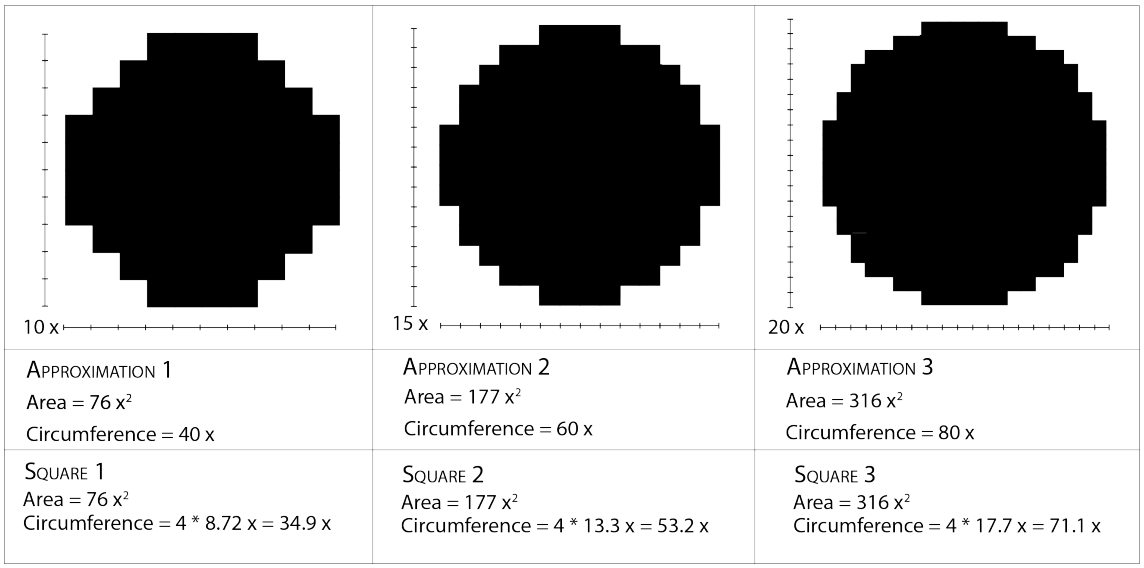


Figure D.1: Area and circumference of three different approximations of a circle. All approximations are compared to a square with equal area as the circle approximations

Even when moving to a near-infinite granulation of the circumference the approximation of a circle is not optimal. This can be graphically proven as illustrated in Figure D.2. In this figure it is shown that part of the approximate circle can be relocated to another location on the surface while reducing the circumference.

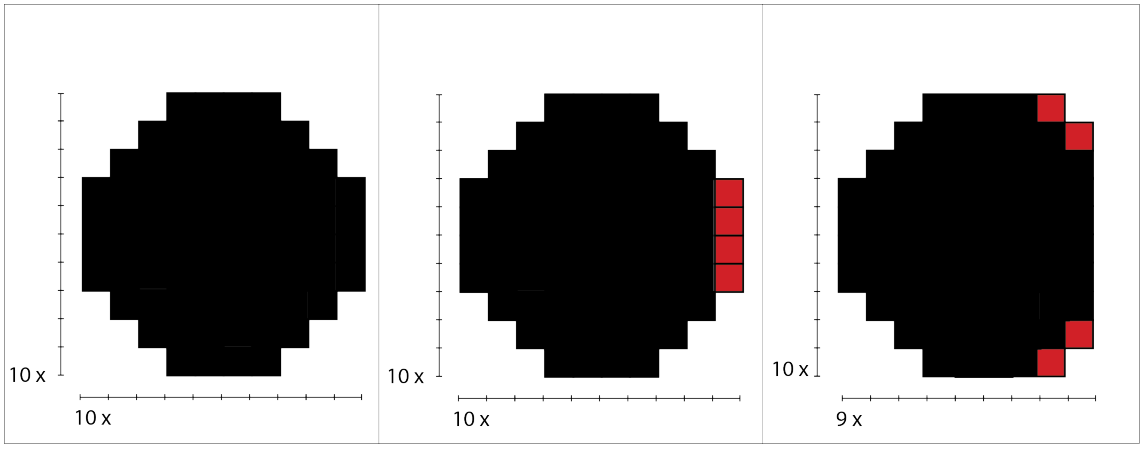


Figure D.2: Graphical transformation of a circle, these steps can be repeated until the approximation is fully transformed into a square.