Multi-disciplinary building optimisation

Boonstra, S.

Award date:
2016

M.Sc. thesis

Multi-Disciplinary Building Optimisation

Technische Universiteit
Eindhoven
University of Technology

Student information:

Name:	S. (Sjonnie) Boonstra
Address:	Doctor Joop den Uylstraat 14-A
ZIP-code:	5612 KW Eindhoven
Phone no.:	(+31)6-18999503
Student-ID no.:	0874727
Date:	19-04-2016

Graduation committee:

Chairman:	dr.ir. H. (Hèrm) Hofmeyer
2nd member:	prof.dr.ir. B. (Bauke) de Vries
3rd member:	prof.dr.ir. A.S.J. (Akke) Suiker
4th member:	dr.ir. A.W.M. (Jos) van Schijndel

M.Sc. thesis

Project:	**Multi-Disciplinary Building Optimisation**
Document title:	**M.Sc. thesis**
Status:	**Final**
Date:	**19-04-2016**

Educational institution

University:	**Eindhoven University of Technology**
Department:	**Built Environment**
Master:	**Structural Design**
Place:	**Eindhoven**

Graduation committee

Chairman:	**dr.ir. H. (Hèrm) Hofmeyer**
2nd member	**prof.dr.ir. B. (Bauke) de Vries**
3rd member	**prof.dr.ir. A.S.J. (Akke) Suiker**
4th member	**dr.ir. A.W.M. (Jos) van Schijndel**

Student information

Student:	**S. (Sjonnie) Boonstra**
Address:	**Doctor Joop den Uylstraat 14-A**
ZIP-code:	**5612 KW Eindhoven**
Phone no.:	**(+31)6-18999503**
Student number:	**0874727**
E-mail:	**s.boonstra@student.tue.nl**

Foreword

Lectori Salutem,

Before you lies the Master of Science thesis 'Multi-Disciplinary Building Optimisation'. This thesis is written as completion to the master Architecture Building and Planning, at the Eindhoven University of Technology.

The subject of the thesis is the enablement of multi-disciplinary optimisation of early building designs. To present day, such optimisation is too elaborate to be performed with existing technology and techniques. However a need for optimal building designs arises due to the impending depletion of natural resources and the pollution of planet earth. Creating methods to optimise early building designs for multiple disciplines within the field today will contribute to the environmentally friendly buildings of the future.

Since May 2015 I have conducted research on the topic, I have learned numerous skills and I was allowed to author and co-author several scientific papers. I am most grateful to all the people who have helped me with my work. The person who has tutored me throughout my graduation is Hèrm Hofmeyer, I am grateful for his time, his answers and his advice. I would also like to thank Jos van Schijndel, Koen van der Blom and Michael Emmerich for tutoring me in their fields of study. A thanks also goes out to all my colleague students, who have given me moral support when I was working besides them. A special thanks goes out to my family who have always been supportive of my work and have always inspired me to continue.

Sjonnie Boonstra,

Eindhoven, April 2016

Table of contents

Annexes:

1. Introduction

1.1. *Subject*

This graduation project is inspired by the project: "*Excellent buildings via forefront MDO. Lowest energy use, optimal spatial and structural performance*", which is funded by the Dutch foundation for technology sciences: STW. The STW-project aims at finding and verifying optimisation techniques that allow spatial modification of buildings to improve structural performance and building physics (BP) performance. The BP-performance is for this project limited to the heat balance of a building, in which heat losses and/or gains need to be minimised. Out of the STW-project two topics are selected for the graduation project, namely the representation of design spaces (all design variables in an optimisation problem) and the selection/development of a building physics analysis tool. This section will discuss the preceding and current research related to the project first, the scope of the graduation project is then discussed and finally the goals for the graduation project are specified.

Preceding and current research

A related Ph.D.-project has been finished preceding this project, it entailed research in which a toolbox was developed that can modify a spatial building design in order to improve its structural performance. This optimisation toolbox can effectively find structurally improved building topologies, however results are one-sided when also considering performance in other disciplines like building physics. To improve the toolbox a collaboration with the Leiden Institute of Advanced computer Sciences has been initiated, this collaboration is committed with professor Michael Emmerich and Ph.D.-student Koen van der Blom. The collaboration aims at adding more disciplines to the toolbox, starting with building physics, and to improve the optimisation techniques used by the toolbox. The current optimisation technique uses a super structure free representation of the design space which is to be defined later, but means that it can search a large (unlimited) domain of solutions. Techniques using a super structure representation of a design space can only search a small and limited domain of solutions, which implies there may be better solutions outside of that domain. The main advantage of techniques that use a super structure representation is that they can employ advanced search algorithms. Therefore these techniques can guarantee to find the best solutions in the search domain (or close to it). A super structure free approach cannot employ these advanced optimisation algorithms, however the search domain is larger and may therefore contain the better solutions. The current research aims at employing the best of both optimisation techniques by alternately using both techniques on one design space.

Scope and goals of the graduation project

As stated, the two topics for this graduation project are the design space for spatial building modification and a building physics analyses tool, more specifically a heat balance calculation. In this report two design space representations are presented one super structure free and one super structured. Thereafter a conversion between design spaces is presented and verified, which in the future will enable the employment of the best of both optimisation techniques by alternately using the different techniques. A discussion is then held which considers some topics around the design spaces of building optimisation. On the topic of design spaces a scientific conference paper was written and a conversion program including a visualisation to convert one design space representation into the other and vice versa was developed. This report will not consider the building optimisation itself nor any implementation of optimisation techniques, only the representations of the design space are considered which forms the foundation of all future optimisations.

The topic on the building physics analyses tool entails an investigation about the selection of a BP-simulation program. This tool should be able to handle the building model in the optimisation toolbox, also it should be possible that the tool can be integrated into the toolbox. The goal is to select such a tool and facilitate the means to integrate it into the toolbox.

1.2. *Optimisation in general*

Design space

Some general knowledge is required when discussing design spaces in optimisation, before discussing optimisation itself it is important to know the definition of a design space in this report. Therefore first design variables must be defined, design variables are all variables that define the design which is subject to optimisation. Arbitrary examples of design variables are: temperature (continuous) when designing a production process, a type of material (discrete) when designing a building's envelope or a pixel's on/off-state (Boolean) when designing pixel art. For design optimisation a selection of design variables is used to generate solutions, the design space contains all solutions that are possible by combining its design variables. A design space is thus a virtual multidimensional space of n-dimensions, where n is the number of design variables. Solutions that are not realistic or even impossible in the real world may be included in design spaces, such solutions can be excluded from optimisation by means of constraints, which is discussed later.

In optimisation different solutions are assessed, an assessment is often performed on a representation of the subjected design. This might result in a loss of control over design variables or the variables itself. Here the design space of a design representation is called a design space representation, this indicates that the design space might be limited compared to the real world design space.

Optimisation

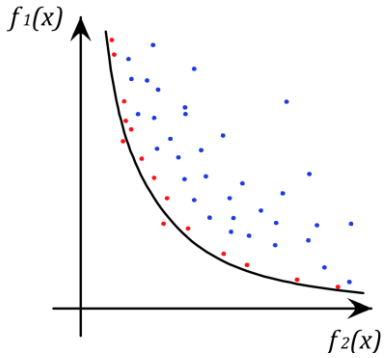
A small introduction into optimisation in general is given, starting with the notation. All optimisation problems are generally denoted in mathematical form as in Expression (1). Here $f(x)$ is called the objective function, it operates on solution x that is obtained from the design space representation. Different solutions are evaluated by the objective function during optimisation, the solution that returns the smallest value to the objective function is called the optimum. The design space representation is delimited by constraints, in the expression $g_j(x)$ are m inequality constraints and $h_k(x)$ are n equality constraints. When a solution x returns false for any of the constraints than that solution may not be used in the evaluation of the objective function.

$$\begin{aligned} &\underset{x}{minimise} && f(x) && \\ &subject\ to && g_j(x) \leq 0 && i = 1,2,\dots,m \\ & && h_k(x) = 0 && j = 1,2,\dots,n \end{aligned} \tag{1}$$

An optimisation may be subjected to multiple objective functions, as presented in Expression (2). Here l objective values exist, each of them operate on solution x. Again different solutions are evaluated during optimisation, now several optima will be found, because one solution is unlikely to be the optimum for all objective functions. In optimisation this is called trade-off, when a solution is evaluated optimal for one objective function then another objective function must "trade" some of its optimality.

$$\begin{aligned} &\underset{x}{minimise} && f_i(x) && i = 1,2,\dots,l \\ &subject\ to && g_j(x) \leq 0 && j = 1,2,\dots,m \\ & && h_k(x) = 0 && k = 1,2,\dots,n \end{aligned} \tag{2}$$

Trade-offs between objective functions are often not discrete changes, but are related with changes in (derivatives of $f(x)$ related to) one or more design variables. These gradual trade-offs can be investigated with Pareto fronts, Figure 1-1 illustrates such a front for two different objective functions. A Pareto front is computed by regarding all non-dominated solutions (red dots in the figure), non-dominated points are defined on the basis of two criteria: A solution x^* is not dominated when there exists no other solution x such that $f_i(x) \leq f_i(x^*)$ for $i \in L\{1,2,\dots,l\}$ and $f_i(x) < f_i(x^*)$ for at least one $i \in L$. This means that there cannot be a solution x that results in a smaller evaluation for all of the objective functions, but that there may be solutions that result in a smaller evaluation for some of the objective functions. In other words a solution is non-dominated if another solution would worsen the evaluation of any objective function. Pareto fronts are useful because they hold information about the design process, this information can for example be interpreted into design rules.



1-1 Evaluation points for objective functions $f_1(x)$ and $f_2(x)$ blue dots are dominated solutions, red dots are non-dominated solutions. The Pareto front is fitted to the non-dominated solutions.

Optimisation problems are treated in a rather mathematical way, for instance the objective functions and constraints are expressed as functions. In a real world optimisation problem it is often difficult, if not impossible, to express all objective functions and constraints in mathematical form. A mathematically unknown function or constraint is called a black box, which means that the returned value is computed by either an algorithm, simulation or even a real world measurement. Optima of problems cannot be found analytically when black boxes are used, specialised search algorithms can then be used. Success of these algorithms is guaranteed when all solutions are evaluated by the objective function, however time becomes decisive when there are many possible solutions. Search algorithms are therefore often specialised in evaluating as few as possible solutions while maintaining a high chance of finding the optimum. The next paragraphs will discuss in more detail how search algorithms work and how a design space representation influences a search algorithm.

Optimisation algorithms

Optimisation algorithms aim to minimise the number of solutions to be assessed to limit computation time. Many different algorithms have been developed over the years, each of which have their own advantages or improvements over others. The most important property of an optimisation algorithm is the employed search technique, which purpose is to find optima in the given design space representation. Some basics of search techniques are discussed first, followed by some examples of distinguished optimisation techniques.

A search is generally classified to be either global or local depending on how solutions are selected for assessment. A global search generally selects solutions without considering previous solution assessments this can be done randomly but also systematically e.g. by defining increments in continues design variables. In a local search previous solution assessments are used to select new solutions, either by means of heuristic operations on solutions like steepest descent and known design rules or by genetic operations like cross-over, mutation, replacement and reproduction. Local searches are prone to finding local optima, which is not the best solution from the design space representation however a better solution cannot be found without considering worse solutions first. A global search is not sensitive to local optima, but it is not checked if there are better solutions near the found optima. Therefore global and local searches are often combined, this way global searches increase the chance to find the approximate location of the global optimum in the represented design space while local searches can find the exact location of that optimum.

Another important factor in optimisation algorithms is how the design space is described by the program. In the simplest case a design is represented with only integers, design solutions are then described by an integer (non-) linear programming approach. An optimisation problem is linear when the objective function and constraints are linearly dependant on changes in the solution's design variables. Design solutions are described by a mixed integer (non-) linear programming approach when design variables are of a type other than integer. The choice of programming approach has great influence on the design space representation (or vice versa) as the chosen approach determines what can and what cannot be described by the design space representation. A specific approach will become part of the optimisation algorithm as it is used to define the solutions that are represented by the design space. Moreover the approach has influence on how global and local searches are performed.

A renowned multi objective genetic optimisation algorithm is NSGA-II (Deb et al. 2002), it is an algorithm that performs both global and local searches. With NSGA-II first a global selection of solutions is initiated, the initial (parent) population, these solutions are assessed and then a part of the population is rejected based

on the extent to which they are dominated. Accordingly the remaining solutions are used for the creation of a new population by using evolutionary operations like reproduction, mutation and crossover. The algorithm is then repeated for a specified number of iterations and leads in most cases to Pareto fronts. Bandaru & Deb (2015) used the NSGA-II algorithm to derive relationships between design variables, these relationships are then tracked throughout the optimisation process i.e. their evolution. Relationships give useful information that may be used in design rules of optimal designs, the evolution of relationships is of interest when hierarchical relationships exist e.g. a design variable does not contribute until a certain condition is met.

Super structures

The generally used super structured design space representation cannot change during optimisation i.e. the existence of design variables is invariable. This means for example that in the design of a production process there are only two temperature states during the whole process or in the design of a façade only one material type can be used for the entire façade or in the design of pixel art only four hundred pixels can be used. Operating on the existence of design variables would either increase or decrease the design space's size. A profuse amount of design variables could be incorporated in the design space when the number of design variables is not known beforehand however this dramatically increases the design space's size. A super structure free design space representation does not fix its design variables i.e. operations like add, delete and replace can be used on them. These operations allow the design space representation to vary in size during optimisation. Design variables can be removed or added when this would have positive effects on the design's performance, this offers solutions to problems where the number of design variables should vary or where it is not known.

The main disadvantage of a super structure free design space representation is that it cannot be handled by most optimisation algorithms. Mainly because super structure free design spaces are an uncommon approach and algorithms are not designed for such tasks. But also because it should be defined how the operators modify the design space with the aim to minimise the objective function, this can be challenging. Examples of super structure free approaches are found more often in literature even though it is still an uncommon approach. Emmerich et al. (2001) for example propose to use operators on design variables in evolutionary design of chemical process networks. Voll et al. (2012) use a super structure free approach by using replacement operators on design variables in the design of an optimal distribution system for energy supply. Baldock & Shea (2006) use a super structure free approach to design an optimal layout of trusses in a structural building frame.

1.3. *Optimisation in the built environment*

Optimisation has been a topic in the built environment for a long time, it is a task traditionally performed by designers and engineers in a trial and error fashion. Knowledge and technology have led to major improvements in building optimisation e.g. math provided an opportunity to analytically optimise structures and computer technology provided an opportunity to simulate building behaviour prior to its realisation. Optimisation itself is a matter of mathematics and computations and has no common ground with the built environment, defining the optimisation problem however requires knowledge about both fields. This si because optimisation is still limited at the time of writing, a complete building design cannot be handled by today's computers as this leads to a too large design space. Designers and engineers are still required in a design process. With knowledge and experience they can represent a design space with sufficient accuracy so that it is small enough to be optimised by today's means and methods. This section discusses some research on building optimisation and more specifically how the design spaces are represented and how optimisation results may be interpreted by researchers

Building physics

Optimisation in the built environment is often limited to single disciplines like structural design, building physics or construction technology most of which also have sub disciplines. Many examples of optimisation in the built environment can be found, some of which are discussed next. Energy performance of a building is a popular topic, Tuhus-Dubrow & Krarti (2010) for example minimise both energy cost and monetary cost by modifying pre-sets of building shape, azimuth, aspect ratios in building shape and properties of several constructions like walls. The design is represented by a genome, where after a genetic algorithm is used to search for optima. Conclusions on building shape are made by assessing the optima found for specific

objective functions at each pre-set shape. The optimal shape of a building with respect to minimal energy costs is a well-known example that can be derived analytically and results in a spherical building. Such analytical optimisation is also performed for multiple objectives by Marks (1997) who derived the relationship of a building's shape with the energy cost and monetary cost. This leads to a system of nonlinear equations which is then numerically solved directly into the Pareto front belonging to the optimisation problem. Another effort to find new building shapes with improved energy performance is presented by Yi & Malkawi (2009), who represent a building's envelope with agent points. These points' positions are modified by a genetic algorithm, an envelope results from the positions of each point in relation to neighbouring points. This way the building's shape is pre-encoded without using pre-sets of shapes. This allows to generate more complex and new insights in optimal building shapes.

Structural design

Structural stiffness is another popular subject for optimisation and research to optimisation dates back to over 50 years, optimal configurations of trusses for example are already analytically investigated in (Chan, 1960). Optimisation of single structural components like beams, columns and floors can be done by structural topology optimisation, which considers material placement in order to maximise structural stiffness. Structural topology optimisation can conveniently use values obtained by Finite Element Analysis (FEA) of structural components to assess and modify elements. Operations like add, delete and replace or gradient element densities can be performed on elements to modify a design. This kind of optimisation can generate optimal designs from which loading and boundary conditions are defined beforehand for example Jang et al. (2010) use it to generate a new design of a flatbed trailers, Liang et al. (2000) use it to design a bracing system for a building frame, Hofmeyer & Davila Delgado (2015) use it to find optimal spatial layouts of buildings, Hammer & Olhoff (2000) use it to optimise structures subjected to pressure loadings, in which it is important to include effects by changing surfaces. Heuristic searches are most often employed when optimising a finite element representation of structural components, mainly because the design space is generally too large for genetic approaches but also because the derivative of strain energy with respect to element densities can be expressed and used for aimed modifications. Heuristics are however not always available, genetic algorithms are for example employed when optimising dimensions of structural components. Kang & Kim (2005) employ a genetic algorithm to modify plate thicknesses in the design of a stiffened plate to minimise weight and maximise post buckling strength. Iuspa & Ruocco (2008) optimise stiffened plates to either minimum weight or maximum buckling loads by means of a genetic algorithm that can vary the type of stiffeners and plate dimensions. Genetic algorithms are also employed when considering the stiffness of a complete building e.g. by Baldock & Shea (2006) who optimise a bracing system for a building frame by using a super structure free approach.

Other examples

Other interesting topics in building optimisation have been found as well. Wright et al. (2002) for example investigate trade-offs between monetary capital and operation costs and user discomfort versus monetary energy costs. An existing building optimisation toolbox called GENE_ARCH is reviewed by Caldas (2008), which is a toolbox that uses a genetic algorithm to optimise program defined design space representations that include for example building and construction dimensions or construction type. Fong et al. (2006) use a genetic algorithm with evolutionary operators to optimise the energy use of an HVAC-system. A state of the art of optimisation in structural design is presented by Kicinger et al. (2005), who studied the field of evolutionary computation in context of structural design. Emmerich et al. (2008) have studied the applicability of several methodologies for optimisation of building physics performances by use of building physics simulation.

1.4. *Building physics Simulation*

Building physics is an established field in the built environment, many studies and education aim ate different topics in the field. One of these topics is computer simulation, in which one or more properties of a building design are simulated over time. Properties like temperature, ventilation, moisture or insolation can be simulated.

The topic of building physics simulation is established, several educational books have been written on the topic, for example the ones by Underwood & Yik (2004), Clarke (2001) or by Hensen & Lamberts (2011). The mentioned books have been written to give an understanding in the workings of most commonly used simulation methods. Scientific research on the other hand focus on e.g. benchmarking of simulation programs, increasing simulation accuracy, increasing simulation speed or the development of new models to simulate novel constructions or systems.

The focus in this work lies on a little demanding simulation method and/or program, since optimisation may require a high number of building physics analysis. It is thus of importance that the selected method is fast and that the selected program does not demand a large amount of building information. When regarding simulation programs there is some research that focuses on user friendliness of programs. For example Attia et al. (2009) have made an overview of programs in which they asses how suitable each program is for an architect's user intentions. User friendliness is however not a popular topic, as mostly experts work with the simulation tools. But user friendliness can be of use when regarding multi-disciplinary design, in which non-experts need to work with simulation tools. Simulation speed and the amount of required data are better researched, under the title of simplified building physics models. This topic in research found its origins in times where computational time was expensive, but has gained renewed interest for applications in optimisation and simulation based design. Van Schijndel & Kramer (2014) have combine three different modelling techniques, each with different levels of detail. The research concludes that all considered techniques give comparable results when low spatial resolution is applied, but that the simplified models are faster. Kramer et al. (2012) give a literature review of simplified thermal and hygric models, clarifying what approaches are applied and what their (dis) advantages are. The level of detail in a building physics model can also be changed as is done by Kramer et al. (2013). In their research it is concluded that the simulation time may be decreased drastically while simulation results remain accurate.

2. Design spaces

This chapter covers the design space representations of building spatial design optimisation. Firstly the boundary conditions for building design are discussed including considerations and constraints for the solutions space. Most important is the conclusion that a large design space is desired for multi-disciplinary building optimisation. Thereafter two representations are presented for use in building optimisation, both are considered for a super structured and super structure free design space. As will become clear one representation is best suited for a super structure free approach and the other is more suitable for a super structured approach. The strengths of both approaches are then proposed to be used in a method to find an optimum both effectively and thoroughly in a large design space. Therefore algorithms that convert between both representations are presented. Finally a discussion is held on some considerations for the topics presented in this chapter.

2.1. *Design boundary conditions*

Building optimisation can be performed with different design space representations all with different levels of detail, the desired properties and detail of a representation are established beforehand to secure optimisation quality. Starting with the requirements of a building model that is to be assessed on structural and building physics performances. Followed by the preferences and constraints of the design space and finally design modifications are discussed.

Requirements

The building model should only have to hold spatial information, which means information about a space's location, shape and dimensions. This causes lack of information to run performance assessments e.g. a structural design including loads is required for structural analysis. These deficiencies of the model can be solved by grammars that generate required building components based on design rules and procedures. A structural grammar could for example determine whether columns or walls are used based on a space's dimensions.

Using grammars thus allows to use a building model with only spatial information, making possible to investigate the influence of a building's topology on its performances. It is also imaginable to use different versions of grammars, different versions of structural grammar could be for example steel structure, wooden slab structure, concrete core with columns etc. When multiple grammars are available it is possible to incorporate grammars in the design space to investigate possible trade-offs between e.g. construction type and topology. A building model for optimisation that contains only spatial information is beneficial as it decreases the search space for the optimisation and thus it results in a better chance of finding the global optimum. Grammars may not result in optimal designs, however grammars could in future work be developed by machine learning on the basis of optimised results for their specific purposes.

Preferences

The goal of every optimisation is to find the global optimum for the problem at hand, global optima are however rarely found and generally cannot be proven to be global. A global optimum in an optimisation problem may not be found mainly because of two reasons, namely the optimisation search gets caught at local optima or the global optimum is not included in the design space representation. The issue with local optima is largely related to the employed search algorithm, algorithms often have advanced functions to escape such local minima as efficient as possible but may still only find better optima rather than the global optimum. The issue in which the design space representation does not include the global optimum is obviously related to the representation itself. It is thus of interest to define it as such that an as large as possible design space is represented or to define as such that the global optimum is included in the representation. As the global optimum is not known a priori it is not possible to purposely include it in the design space representation, which means the represented design space should be defined as large as possible. However it becomes harder to search a design space when it becomes larger, as either more solutions must be evaluated or an optimal solution must be determined with relatively less evaluations. This problem is also encountered in topology optimisation of building designs, a variation of a few spaces can already lead to a large number of solutions as several spatial configurations of the spaces are to be combined with an optimisation of their dimensions.

A large represented design space is preferred for spatial building optimisation to increase the chance of finding a better or the global optimum, this means that large topology variations are desired. Long optimisation times should however be avoided, this can be done by constraining the space as will be discussed in the following section. Another method is to employ an effective search that can quickly move through the search space based on simple modification rules (heuristics) that increase the design's performance. This method is however a local search and thus prone to locking into local optima, therefore a method should be developed to escape local optima.

Constraints

The search space and thus optimisation time can be reduced by constraining the search space. Constraints may be obvious like invalid solutions as floating structures and overlaps. Practical constrains can also be defined e.g. all spaces are cuboid, which simplifies the representation of a building design and also reduces optimisation time. All constraints that will be applied to the optimisation problem at hand are discussed in this section.

Overlap
Overlaps are not allowed during the optimisation of a building as an overlap will cause problems in the design model/process.

Connectedness
Spaces must be linked to all other spaces by a network of connected spaces, separated (groups of) spaces would introduce new buildings while the model describes one building.

Cuboid
Spaces must be cuboid, this constraint simplifies the representation of a building model. It is practical in terms of saving optimisation time but also in terms of building use as orthogonal floor plans are generally accepted. It should however not be considered to be a

Ground connection and aboveground
A building should be connected with the ground with at least one space to ensure access to the building and to ensure the structural design is founded. Besides that, all spaces must be located above ground to limit the extensiveness of the BPS- and SD- analyses and grammars.

Constant volume and number of spaces
The building's volume and number of spaces are defined at the start of the optimisation and must be maintained during the optimisation process. This allows a building to be designed for a purpose or to meet client demands

Design variation and modification

Modification or variation of a spatial model that only contains spaces entails that only spaces can be modified, as such an optimisation's design space consists out of the used representation of the building's spaces. As stated a space is defined by location, dimension and shape, though the shape is constrained to cuboids and as such only the location and dimension remain to be modified.

The implications of a super structured or super structure free approach should be considered as well when considering design spaces. In a super structured approach there may not be any operations on the design variables itself, in other words only the space location and dimensions can be varied. Operators are allowed in a super structure free approach, thus allowing locations, dimensions or both to be deleted or created. Doing such operations on only location or dimensions will result in invalid spaces depending on the used representation, therefore these operators can generally only work on both variables at the same time i.e. the entire space.

A building optimisation's design space is defined by the used building representation, generally it is super structure free when spaces can either be created or deleted. However the volume and number of spaces are constrained, thus using a super structure free approach does not seem appealing. However when looking at heuristics for building optimisation it turns out that removal and creation of spaces can be powerful operations to quickly increase a design's performance inside a large design space, as is shown by Hofmeyer & Davila Delgado (2015).

Two representations are presented in the following section, one is suitable for a super structure free approach the other for a super structured approach. The free approach may thus be suitable for using heuristics to find optima in a large search space quickly and effectively, at the cost of locking into local optima. The super structured approach of the design space can result in the best optima found in a small search space, at the cost of optimisation speed and excluding possible better optima.

2.2. *'Movable and Sizable' representation*

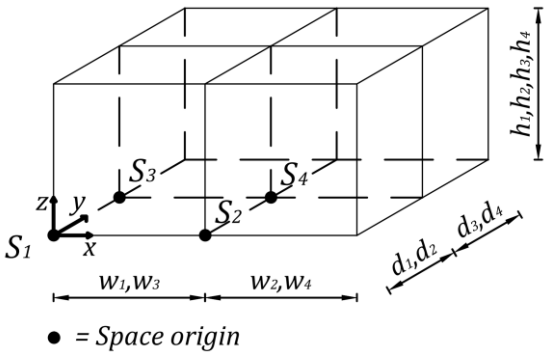
Building representation

The 'Movable and Sizable' representation (MS-representation) is a very direct way of representing spaces, it uses the bare minimum to specify cuboid spaces. This bare minimum consists out of a sets of coordinates (x, y, z) and a set of dimensions (w, d, h) width, depth and height respectively. This representation can formally be stated with the equations in Expression (3).

$$\begin{aligned} &\vec{S} = \left\{S_1, S_2, \dots, S_{N_{spaces}}\right\} \\ &S_i = \{C, D\} \\ &C = \{x, y, z\} \\ &D = \{w, d, h\} \end{aligned} \tag{3}$$

Coordinates and dimensions of a space should be variable when it is desired to optimise the topology of a building using the above representation, otherwise either relocation or resizing would not be possible. Therefore it can be said that the design space consists out of every represented space and thus is represented by vector S. An example of a building composed of four spaces in the MS-representation is given in Figure 2-1.

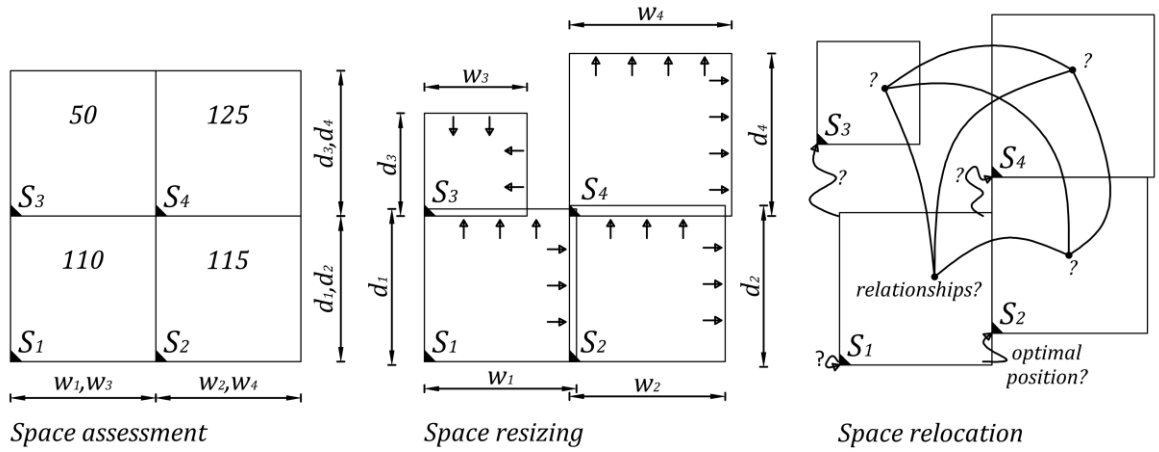
$S_1\{\{x_1, y_1, z_1\}, \{w_1, d_1, h_1\}\}$
$S_2\{\{x_2, y_2, z_2\}, \{w_2, d_2, h_2\}\}$
$S_3\{\{x_3, y_3, z_3\}, \{w_3, d_3, h_3\}\}$
$S_4\{\{x_4, y_4, z_4\}, \{w_4, d_4, h_4\}\}$



2-1 'Movable and Sizable' representation of a building

Super structured approach

The MS-representation is not suitable for a super structured approach, the problem lies within constraining the design space representation. The representation defines only individual spaces without any additional information e.g. a space's relevant position in the considered building is not known nor the relationships with other spaces. This makes it hard to modify the building design, as every space then needs constraints in relation to other spaces to prevent overlaps but also gaps in between spaces. To define such constraints it is required to describe relations between spaces but also their movement paths as is illustrated in Figure 2-2.



2-2 Arising problems during design modification in the super structured method with the 'Movable and Sizable' building representation

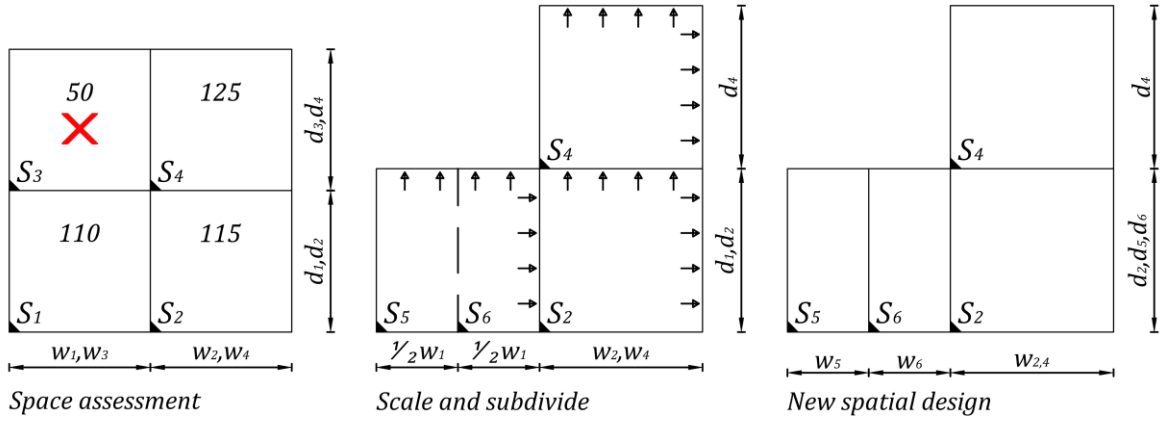
In the depicted example local search is illustrated starting with the assessment of an initial solution, here given by some arbitrary numbers where high numbers indicate high performance. A new solution could be obtained by first resizing all spaces according to their performance while keeping the total volume constant i.e. a space with average score keeps the same size, bad score reduces in size and high score increase in size. Overlaps and gaps appear immediately after resizing, this needs to be solved by relocating spaces to positons where such infringements do not occur, which is the stage where the problems appear. For every movement a space undergoes there are three aspects to be checked:

- **Clashes**, does the space have overlaps with other spaces?
- **Optimal movement**, to which direction should the space be relocated?
- **Relationships**, to what other spaces should the space be related/connected?

Describing this in an algorithm is already challenging, when realising that the movement of every space affects other spaces it may well be impossible to modify designs via this approach.

Super structure free approach

The super structure free approach gives opportunities to modify buildings that use the MS-representation, as now spaces may be deleted and new spaces may be defined during the optimisation process. These operators are used in the existing building optimisation toolbox, as is illustrated in the example in Figure 2-3.



2-3 Design modification in the super structure free method with the 'Movable and Sizable' building representation

Design modification is performed on the basis of the same performance assessment, now one or multiple spaces are deleted from the design space. The building now has too few spaces and too low volume, therefore the entire building is scaled up to the initial volume and one (arbitrary) space are divided into two new spaces. This design modification can formally be represented by Expressions (4) to (7).

Deletion: $$\vec{S}\{S_1, S_2, S_3, S_4\} \rightarrow \vec{S}\{S_1, S_2, S_4\} \quad (4)$$

Scaling: $$\begin{pmatrix} S_1\{\{x,y\},\{w,d\}\}, S_2\{\{x,y\},\{w,d\}\}, \\ S_4\{\{x,y\},\{w,d\}\} \end{pmatrix} \cdot \sqrt{\frac{V_0}{V}} \quad (5)$$

Division: $$S_1\{\{x_1,y_1,z_1\},\{w_1,d_1,h_1\}\} \rightarrow \begin{cases} S_5\left\{\{x_1,y_1,z_1\},\left\{\frac{1}{2}w_1,d_1,h_1\right\}\right\}, \\ S_6\left\{\left\{x_1+\frac{1}{2}w_1,0,0\right\},\left\{\frac{1}{2}w_1,d_1,h_1\right\}\right\} \end{cases} \quad (6)$$

New Design: $$S\{S_2, S_4, S_5, S_6\} \quad (7)$$

Using this approach, it is not needed to check constraints as no overlaps are created during modification and relations between spaces are only created where they already existed in the original space. If a building is therefore correctly entered by the user there is no need to check constraints. Here the super structure free approach is used, not to vary in number of design variables, but to prevent elaborate constraint checking.

2.3. *'Super Cube' representation*

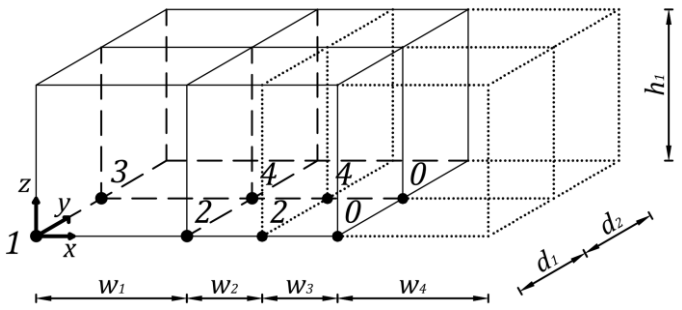
Building representation

The 'Super Cube' representation (SC-representation), by M.T.M Emmerich (LIACS), uses four vectors to describe a building: $B(\overrightarrow{w_i}, \overrightarrow{d_j}, \overrightarrow{h_k}, \vec{b}^{\,l}_{i,j,k})$, together they represent multiple cells inside a large cuboid that hold information about the spaces of a building. Expression (8) shows the variables used, here $b^l_{i,j,k}$ describes the existence of a cell with indices i, j and k in space l, where a value "1" means the cell is active and describes part of space l while "0" means the cell is inactive. Following this, i is the x-, j the y- and k the z-index of a cell, while l is the space index. Finally, w_i, d_j, h_k describe the continuous dimensioning of the super cube's cells. The entire super cube is used to perform design modifications, therefore the complete design space is described by the vectors $\overrightarrow{w_i}, \overrightarrow{d_j}, \overrightarrow{h_k}$ and $\vec{b}^{\,l}_{i,j,k}$.

$$
\begin{aligned}
&i \in \{1,2,\dots,N_w\} \qquad w_i \in \mathbb{R} \geq 0\\
&j \in \{1,2,\dots,N_d\} \qquad d_j \in \mathbb{R} \geq 0\\
&k \in \{1,2,\dots,N_h\} \qquad h_k \in \mathbb{R} \geq 0\\
&l \in \{1,2,\dots,N_{spaces}\} \qquad b^l_{i,j,k} = \begin{cases} 1 & if\ cell\ (i,j,k)\ belongs\ to\ space\ l \\ 0 & otherwise \end{cases}
\end{aligned}
\tag{8}
$$

The design space consists of all four vectors used in the SC-representation, as these are all needed if a space's dimensions and location are to be modified. Figure 2-4 illustrates the SC-representation in which a building of four spaces is described, each space by one or more cells. It is also possible that cells are not used in the description of a building, such cells are still part of the super cube but not of the building. The super cube itself thus does not embody a building but is merely a virtual representation of a building.

$\vec{w}\{w_1, w_2, w_3, w_4\}$
$\vec{d}\{d_1, d_2\}$
$\vec{h}\{h_1\}$
$\overrightarrow{b^l_{i,j,k}}\{b^1_{i,j,k}, b^2_{i,j,k}, b^3_{i,j,k}, b^4_{i,j,k}\}$
$b^1_{i,j,k}\{1,0,0,0,0,0,0,0\}$
$b^2_{i,j,k}\{0,0,1,0,1,0,0,0\}$
$b^3_{i,j,k}\{0,1,0,0,0,0,0,0\}$
$b^4_{i,j,k}\{0,0,0,1,0,1,0,0\}$



2-4 'Super Structured' representation of a building

Super structured approach

Using a super structured approach of a design space with the SC-representation is more feasible than for the MS-representation. An advantage of the SC-representation is that the spaces' locations can be pre-encoded in the vectors, thus reducing the search space significantly. The main advantage is however that this notation allows to check constraints by using mathematical equations i.e. conditions that must be met. Mixed integer non-linear programming (MINLP) techniques can then be used to generate designs that meet constraints.

In this paragraph the conditions for each of the constraints are presented, the conditions are formulated by Ph.D.-student Koen van der Blom, who at the time of writing develops the mixed integer non-linear programming and optimisation techniques for the SC-representation.

Condition 1 – non overlap

Every space is represented by a separate bit mask of the entire super cube ($b^l_{i,j,k}$), therefore overlap can occur. The first condition, Equation (9), checks this by summing up the value of a cell in each mask, when the sum is smaller than or equal to one there does not exist an overlap at the position represented by that cell.

$$\forall_l : \forall_{i,j,k} \sum_{l=1}^{N_{spaces}} b_{i,j,k}^{l} \leq 1 \tag{9}$$

Condition 2 – cuboid spaces

This condition checks if all cells that are assigned to a space together form a cuboid. This is done by first building a layer of "zero" cells around the super cube as is described by equation (10).

$$\begin{gathered}\forall_l : \forall_{i,j,k} \in \{0, \dots, N_w + 1\} \times \{0, \dots, N_d + 1\} \times \{0, \dots, N_h + 1\}: \\ i = 0 \vee j = 0 \vee k = 0 \vee i = N_w + 1 \vee j = N_d + 1 \vee k = N_h + 1 \Rightarrow b_{i,j,k}^{l} = 0\end{gathered} \tag{10}$$

Accordingly the super cube is iterated per space for each i-, j-, and k-index in search for changes between "zero" and "one" cells. For each direction searched, it should hold that when changes do occur then they should occur at the same indices. Equation (11) shows how this check is performed in the z-direction. Note that this constraint only accounts for a cuboid shape, and that internal voids may still be present in a cuboid space. Condition 3 is introduced to account for such voids by checking if a space is orthogonally convex.

$$\begin{gathered}\forall_{\boldsymbol{l}}: \\ \forall_{i_1,j_1,i_2,j_2:} \left(\left(\sum_{k=1}^{N_h} k\left(1 - b_{i_1,j_1,k-1}^{l}\right) b_{i_1,j_1,k}^{l} \right) - \left(\sum_{k=1}^{N_h} k\left(1 - b_{i_2,j_2,k-1}^{l}\right) b_{i_2,j_2,k}^{l} \right) \right) * \\ \left(\sum_{k=1}^{N_h} b_{i_1,j_1,k}^{l} \right) \left(\sum_{k=1}^{N_h} b_{i_2,j_2,k}^{l} \right) = 0 \\ \forall_{i_1,j_1,i_2,j_2:} \left(\left(\sum_{k=1}^{N_h} k b_{i_1,j_1,k}^{l} \left(1 - b_{i_1,j_1,k+1}^{l}\right) \right) \right. \\ \left. - \left(\sum_{k=1}^{N_h} k b_{i_2,j_2,k}^{l} \left(1 - b_{i_2,j_2,k+1}^{l}\right) \right) \right) \left(\sum_{k=1}^{N_h} b_{i_1,j_1,k}^{l} \right) \left(\sum_{k=1}^{N_h} b_{i_2,j_2,k}^{l} \right) = 0\end{gathered} \tag{11}$$

Condition 3 – ortho convexity of spaces

This condition checks for each space if there are more than changes from zero to one in the super cube. If this is true, then there exists a void in the space, the condition is checked by equation (12).

$$\begin{gathered}\forall_l : \\ \forall_{i,j}: \sum_{k=0}^{N_h} (1 - b_{i,j,k}^{l}) b_{i,j,k+1}^{l} \leq 1 \qquad \forall_{i,k}: \sum_{j=0}^{N_d} (1 - b_{i,j,k}^{l}) b_{i,j+1,k}^{l} \leq 1 \\ \forall_{j,k}: \sum_{i=0}^{N_w} (1 - b_{i,j,k}^{l}) b_{i+1,j,k}^{l} \leq 1\end{gathered} \tag{12}$$

Condition 4 – no overhang

This condition is can be used to make sure all spaces are linked to each other, which means all spaces together form one building rather than a separated clusters of building spaces. This condition can be easily performed by checking for changes of “zero” cells to “one” cells in positive z-direction in the super cube, similar to equation (12). Such a change signifies that there is no cell beneath the considered cell, and thus that there exists an overhang in the building.

Super structure free approach

The super structure free approach is less suitable for the SC-representation, as an operation on a dimension variable (w_i, d_j or h_k) will have impact on the vector describing the spaces ($b^l_{i,j,k}$). Operations on spaces on their turn have consequences for the dimension vectors. These dependencies require to capture and anticipate the consequences of an operation beforehand, which results in an elaborate implementation of simple heuristic rules. The MS-representation is better suited for a super structure free approach, because using the SC-representation would unnecessarily complicate the problem.

2.4. *Conversion between design spaces*

The two presented representations each have their strengths and weaknesses. The MS-representation can be used to quickly search more optimal solutions in a large search space, however it may move in only one direction or it may lock into a local optimum. The SC-representation can be used to thoroughly search the search space and has a high chance on finding the best optima in that search space. Such optimisation however requires a large amount of computation time, therefore the search space should be kept small.

An interaction between representations is suggested to get the best of both approaches. The MS-representation should be used to quickly obtain an improved design after which the SC-representation should be used to refine that improved design with an advanced search technique that operates on a smaller search space. To achieve such interaction it is required to convert the MS- into the SC-representation and vice versa, the algorithms that enable such conversions are presented in this section.

'Movable and Sizable' to 'Super Cube'

This conversion can be divided into two stages, at the first stage the super cube is constructed and at the second stage each space is assigned its cells. The first stage is easily described by the following four steps:

1. ***Find all start and end points of each space in every direction***. For each direction a vector will be initiated that is then for each space 'i' filled with the following:
   - in x-direction: x_i & $x_i + w_i$
   - in y-direction: y_i & $y_i + d_i$
   - in z-direction: z_i & $z_i + h_i$
2. ***Find all unique values***. All unique values are removed from each of the three vectors obtained under step 1.
3. ***Order values***. All values in the three vectors resulting from step 2 are ordered in ascending order.
4. ***Determine the w, d, h values for the super cube from the ordered unique x-, y,- and z-value vectors***. The increments between values when moving through the ordered unique and ordered vectors yield each w, d or h value of the super cube, when maintaining the order in which they are obtained this results in the vectors $\vec{w}$, $\vec{d}$, $\vec{h}$.

The vectors containing the super cube's dimensions are computed in the first stage, in the second stage only vector $\overrightarrow{b^l_{i,j,k}}$ remains to be determined. This vector holds for each space l information whether a cell in the super cube belongs to the space, therefore each cell has to be checked for each space. Checking whether a cell i, j, k belongs to space l is done as follows:

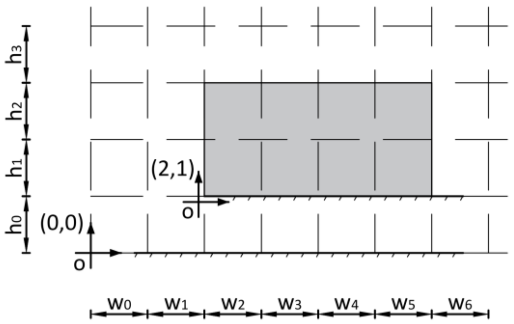
1. ***Obtain the cell's x-, y-, and z-coordinates.*** These are obtained from the ordered and unique x-, y- and z-value vectors from stage one. Each cell's index corresponds with its coordinate in the corresponding vector i.e. the n^{th} element of the x-vector contains the x-coordinate of all cells indexed with $i = n$.
2. ***Check cell's coordinates.*** Check if the cell's coordinates are within the corresponding MS-space's bounds, e.g. for x direction: $x_{room} \leq x_{cell} < x_{room} + w_{room}$
3. ***Assign cell to space*.** Write a '1' for a cell's index when that cell is within the MS-space's bounds in all direction.

Doing the above steps for all cells for each space will transcribe the building from the MS-representation to the SC-representation, the algorithm has then successfully completed the conversion.

'Super Cube' to 'Movable and Sizable'

The super cube does not have any coordinates, the corner cell with indices 0 could be chosen to hold the origin however in a case where cells at the cube bottom are not used this can result in an MS-representation in which the building is floating, see Figure 2-5. Therefore the MS-origin will be initialized to the lowest indices which still contain a cell that describes a space.

3	0	0	0	0	0	0	0	
2	0	0	1	1	1	1	0	
1	0	0	1	1	1	1	0	
0	0	0	0	0	0	0	0	
k / i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	



2-5 Origin of the MS-representation here belongs to cell with index (2,1) and not to (0,0)

The algorithm thus starts by finding the cell that contains the origin, it does this by first assuming the cell with the largest indices contains the origin. Accordingly the indices of each cell that describes a space are compared to the assumed origin cell's indices, an assumed origin index is updated to the value of a considered index if it is smaller. Doing so for every cell describing a space will result in the three lowest indices which are used for a cell that describes a space, these three indices describe the cell containing the origin of the MS-representation.

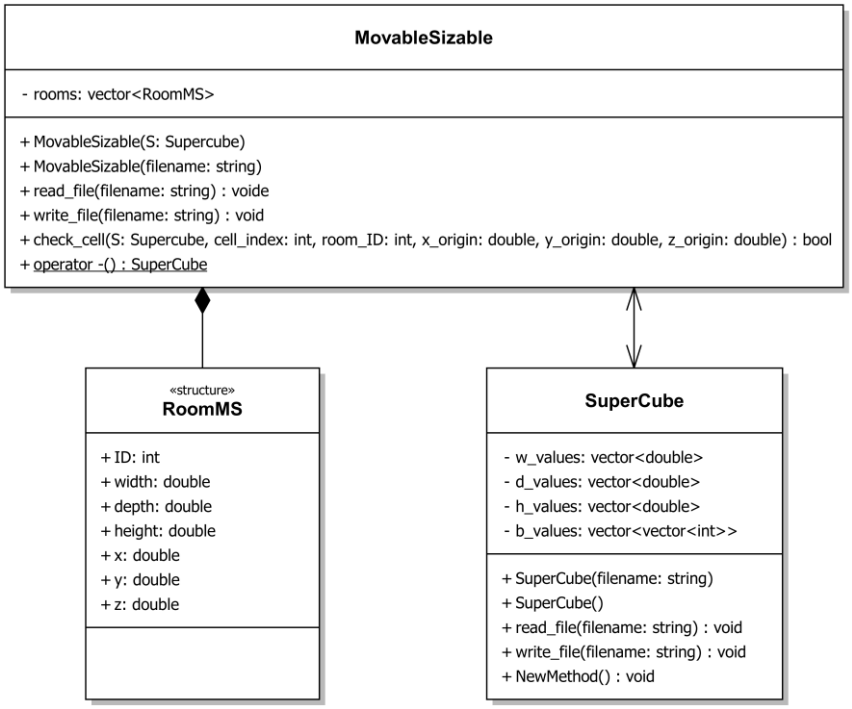
Each vector $\overrightarrow{b^l_{i,j,k}}$ can be transcribed to a space in the MS-representation after the origin has been computed, this is done in the following two steps:

1. ***Find the smallest (min) and largest (max) of indices i,j and k of a space.*** These indices describe all of the outmost cells in the space, when assuming spaces are cuboid.
2. ***Compute the space's coordinates and dimensions in each direction.*** These are computed as follows:
   - in x-direction: $x = \sum_{i=origin}^{i=min-1} w_i$ and $w = \sum_{i=min}^{i=max} w_i$
   - in y-direction: $y = \sum_{j=origin}^{j=min-1} d_j$ and $d = \sum_{j=min}^{j=max} d_j$
   - in z-direction: $z = \sum_{k=origin}^{k=min-1} h_k$ and $h = \sum_{k=min}^{k=max} h_k$

Doing the above two steps for each vector $\overrightarrow{b^l_{i,j,k}}$ will transcribe an SC-representation to an MS-representation, the algorithm has then successfully completed the conversion.

Implementation of the algorithms into C++

The above two algorithms have been implemented in object oriented C++-code, see Annex 1. The code uses a class for each of the representations named 'MovableSizable' (MS-class) and 'SuperCube' (SC-class), see Figure 2-6.



2-6 UML class diagram of the representation classes, put() and get() functions have been omitted for clarity

The conversion from an SC- to an MS- representation is conducted by passing an object of the SC-class as an argument to the constructor of the MS-class. Conversion from the MS- to the SC-representation is done via overloading of the conversion operator. This makes it possible to execute each conversion in simple syntax as demonstrated in Figure 2-7 below.

```
//  this initialises an object (M1) of the MS-class by reading a text file:
    MovableSizable M1("MS_input_file.txt");
//  this initialises an object (S1) of the sc-class by reading a text file:
    MovableSizable S1("SC_input_file.txt");
//  this initialises an object (M1) of the MS-class by converting an SC-object (S1):
    MovableSizable M1 = S1;
//  this initialises an object (S1) of the SC-class by converting an MS-object (M1):
    SuperCube S1 = M1;
//  this converts an existing SC-object (S1) to an existing MS-obect (M1) (via constructor):
    M1 = S1;
//  this converts an existing MS-object (M1) to an existing SC-object (S1) (via conversion overloading):
    S1 = M1;
```

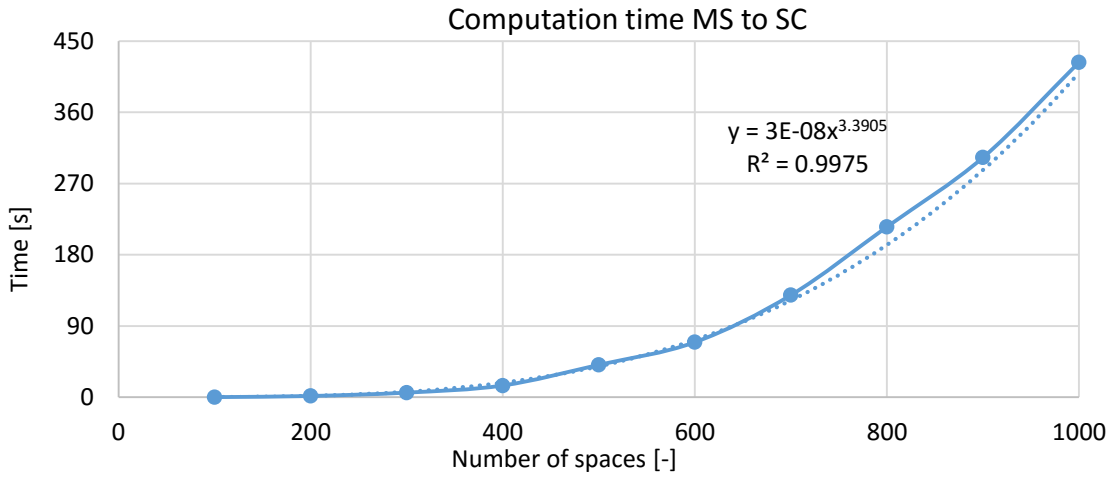
2-7 Example syntax for the use of the representation classes.

This syntax is easy to understand and remember, it provides the opportunity to convert between the two classes without calling functions that are easy to forget. Use of two representations can thus be easily incorporated into the toolbox, as switching between either of the representations is provided. Conversion of representations thus is now possible and gives the ability to switch between optimisation algorithms.

Conversion can also be convenient when a grammar is available for only one representation. For example the structural grammar can at the time of writing only generate a structural design for an MS-representation, conversion prevents writing and maintaining two grammars if the structural performance is desired from an SC-representation.

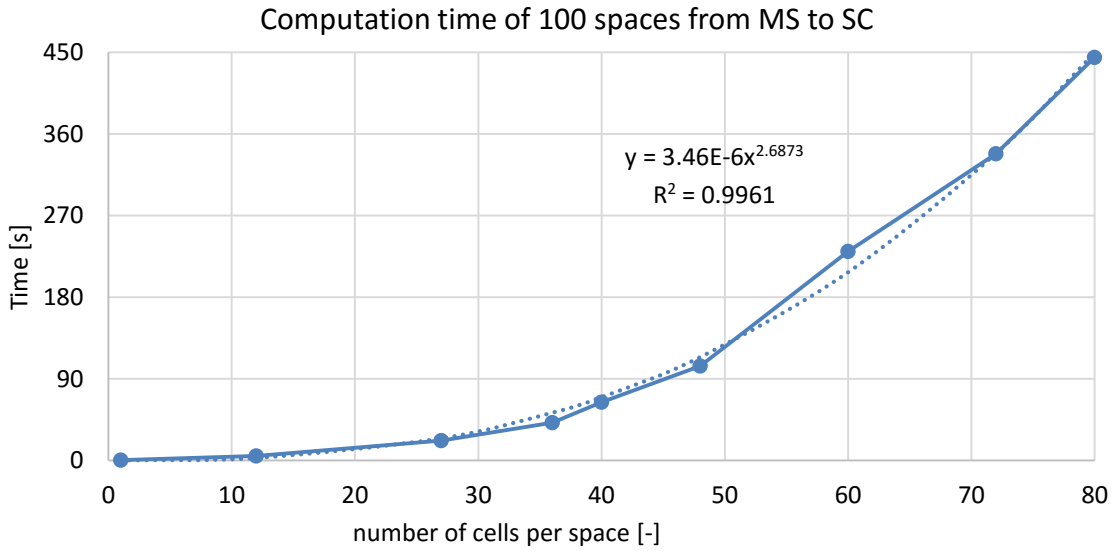
Performance of implementation

As stated in the previous section it can be convenient to switch between representations, however this could be costly in terms of computational time. Therefore the process time of each conversion algorithm for an increasing number of spaces is measured to investigate the costs in terms of computational time, see figure 2-8 for the results for the MS- to SC-representation algorithm.



2-8 Computational time of the algorithm plotted against the number of spaces to be converted

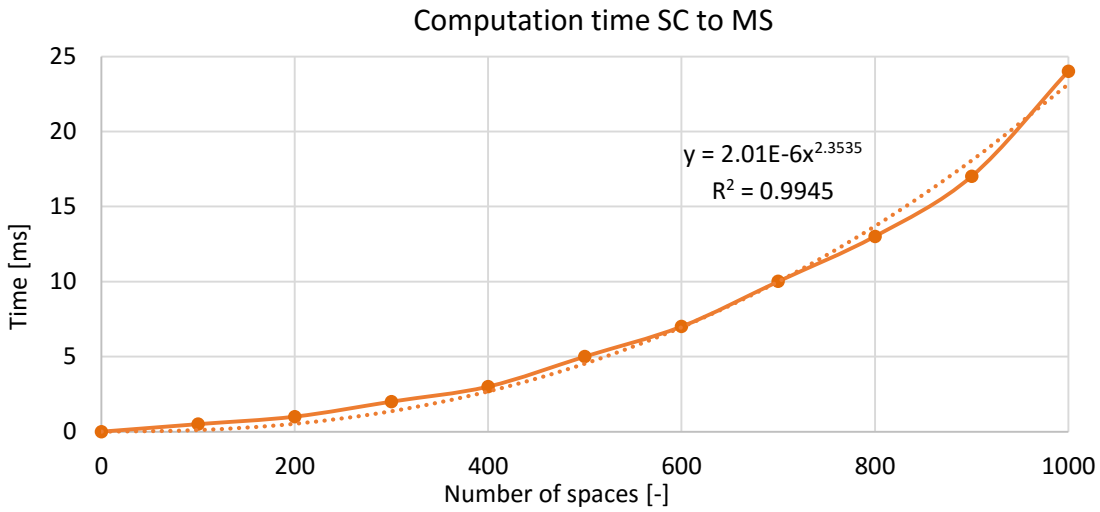
These results are obtained by only using one cell to represent a space in the SC-representation, multiple cells representing one space does however not affect the computing speed as all cells are checked for each space. It would affect the computation time, when it also affects the number of cells to be created e.g. when 100 spaces do not generate 100 cells but 200 cells. This is investigated as well by measuring the time against the number of cells per space each time converting one hundred spaces, the results are plotted in the graph in figure 2-9.



2-9 Computational time of the MS to SC-algorithm plotted against the number of cells per space when one hundred spaces are converted

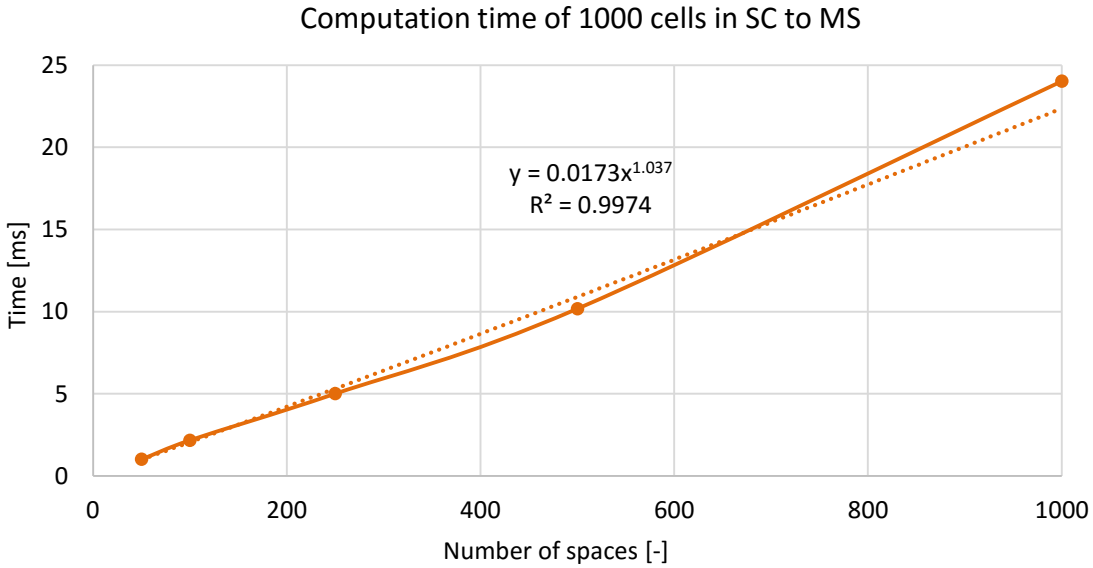
The results show that the computation time of the MS- to SC-conversion is more than cubically dependant on the number of spaces. The computation time is almost cubically dependant on the number of cells per space that are created. It is however not expected that the dependencies enforce each other as during an optimisation the number of cells per space is expected to be constant and independent of the number of spaces, and is at worst expected to show a slight increase when the number of spaces increases.

The conversion algorithm to transcribe the SC- to the MS-representation has been measured in a similar way. The number of spaces is again each time incremented by a hundred spaces, in which each space is represented by only one cell, see Figure 2-10 for the results.



2-10 Computational speed of the algorithm plotted against number of spaces to be converted with one space represented by one cell.

From the results it can be concluded that the conversion from SC- to MS-representation is significantly faster, however again it can be questioned what the influence of multiple cells representing one space may be. Therefore conversions of one thousand cells representing different numbers of spaces are performed, the results are shown in Figure 2-11.



2-11 Computational speed of the algorithm plotted against the number of spaces to be converted with number of cells constant at one thousand.

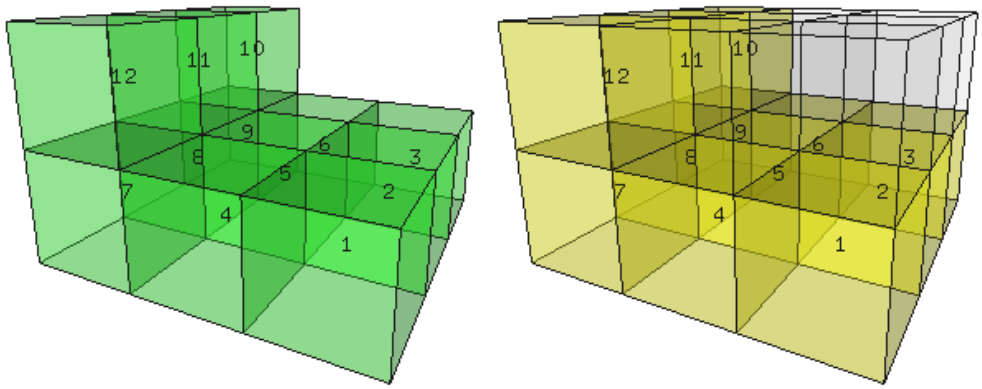
The results show that the computation time is linearly dependant on the number of spaces to be converted from SC- to MS-representation. This is however under the assumption that the number of cells remains constant, if the number of cells increases in ratio to the number of spaces then there is a more than quadratic dependency on the number of spaces.

The SC-to MS-algorithm is faster than the MS- to SC-algorithm which is convenient when this algorithm is used for grammars that only operate on MS-representations as stated before. It can be concluded that the total time of the conversions stays within the order of minutes with the MS- to SC-algorithm and in the order of seconds with the SC- to MS-representation if no more than a thousand spaces are to be converted.

Visualisation

A visualisation for each representation is desirable during use of the optimisation toolbox e.g. to check input or to interpret optimisation results. The visualisation makes it possible to assess the values of a representation in a short amount of time. A C++-code for visualisation is already available from the existing toolbox, the code is based on OpenGL with GLUT which can be compiled across platforms. The visualisation code is built up of 3 modules namely the BSP-module, the model module and the utility module. The utility module contains all GLUT related syntax and holds all classes for vertexes, lines, polygons, labels and colours. The model module contains all model related objects, most importantly a general model class from which specialised model classes are inherited. The general model class holds all information about the model's spaces and space types, per space a list is created that contains sets of vertexes (eight per space). Each set of vertexes is then translated into edges that are then translated into polygons, which are the objects that are eventually rendered. The BSP-module contains classes that incorporate the Binary Space Modelling technique to determine drawing orders on the basis of the camera position. This way the spaces of a model will be rendered in the right order i.e. from the back to the front. The BSP-module also contains the classes that actually draw and render the model in a window, besides another class that determines the camera position on basis of clicking and dragging of the mouse.

The visualisation code can be used for the representations at hand, as the representations only contain cuboid spaces (or cells) that need to be labelled with an identification (or not). For each representation class (MS and SC) this is done by inheriting a new class from the general visualisation model class and adding a constructor that requires a representation class as an argument. The constructors of the inherited classes then translate each space of a representation into the corresponding vertexes. The inherited class also holds information about used colour, transparency or text. The code of the inherited classes can be found in Annex 2 and Annex 3, an example result of the visualisation classes is given in Figure 2-12.



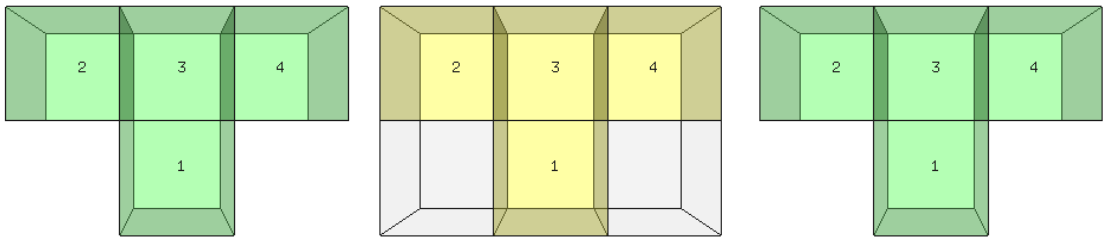
2-12 Left a visualisation of an MS-representation. Right a visualisation of an SC-representation

2.5. *Verification of conversion algorithms*

The functionality of the conversion algorithms are verified in this section. It is expected that the conversions are robust, there should not be any considerations in the creation of a representation in relation with the conversion.

Structures with overhang

As mentioned, there may be a constraint in SC-representation that disallows overhangs because these complicate the checking of the connectedness of spaces. It is thus of interest to see what happens when an overhang does appear in the MS-representation, a T-shaped building consisting of four spaces is therefore subjected to conversion, see Figure 2-13.

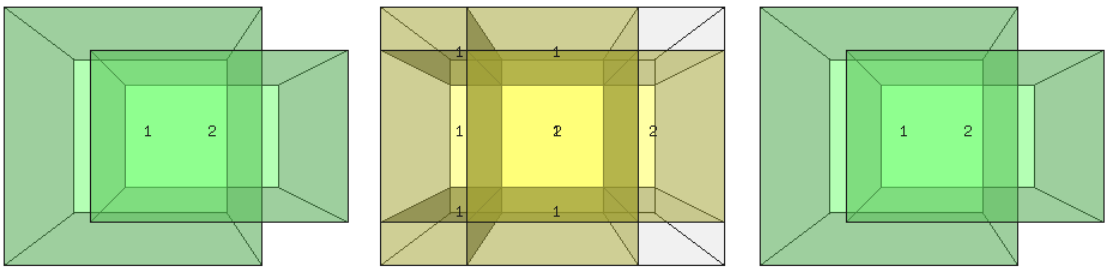


2-13 Conversion of a building with overhanging spaces, from left to right the initial MS-representation is converted to SC-representation which is then converted back again.

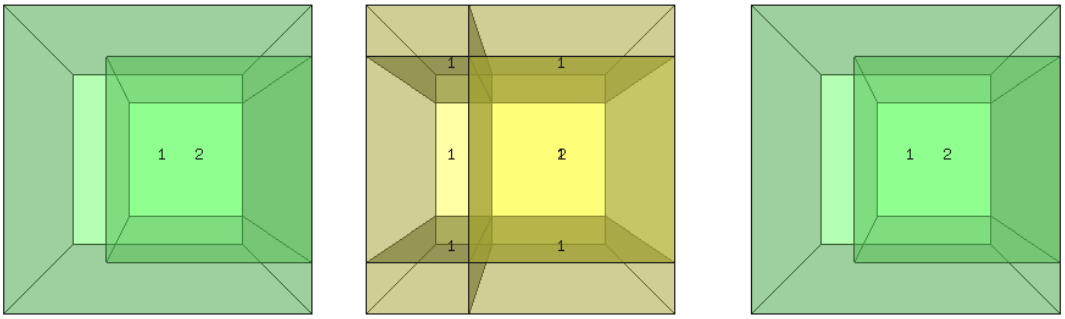
It can be concluded that the conversion algorithms are robust in this aspect, as each building model is exactly identical. This was expected because there are no explicit functions in the algorithms that prevent spaces from being transcribed to another model. If this were to be implemented for spaces that create an overhang it would cause deficiencies in the constrained number of spaces and volume in the building, therefore transcribing an overhang from one representation to another will be allowed. How this problem will be addressed is not yet determined and will require some experience with the optimisations, but it can be considered to switch the connectedness constraints off, which might lead to separate building blocks in the model.

Structures with overlap

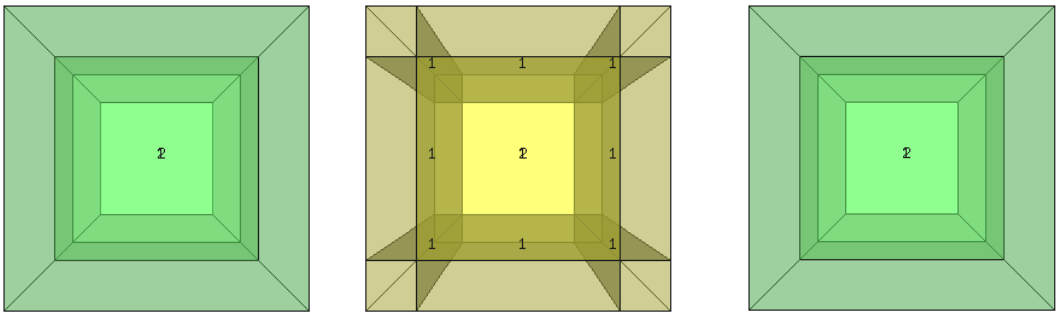
As overlaps are prohibited during optimisation it is of interest how the conversion algorithms handle overlaps. This is investigated by different types of overlap, the results are shown in Figures 2-14 to 2-17.



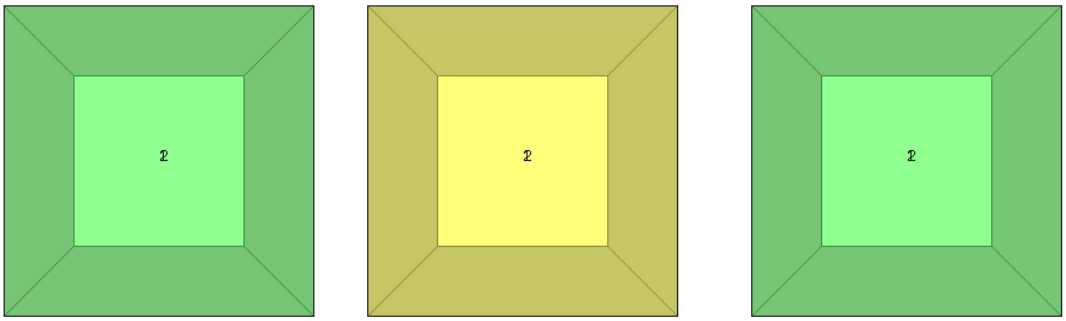
2-14 Conversion of a building with partly overlapping spaces, from left to right the initial MS-representation is converted to SC-representation which is then converted back again.



2-15 Conversion of a building with completely overlapping spaces and some coincident surfaces, from left to right the initial MS-representation is converted to SC-representation which is then converted back again.



2-16 Conversion of a building with completely overlapping spaces without coincident surfaces, from left to right the initial MS-representation is converted to SC-representation which is then converted back again.

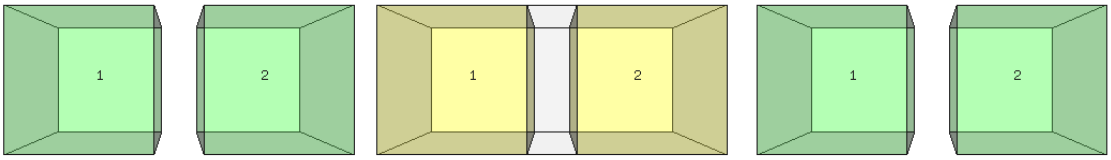


2-17 Conversion of a building with completely overlapping spaces without coincident sides, from left to right the initial MS-representation is converted to SC-representation which is then converted back again.

The results show that the conversion algorithms do not prevent overlaps, on the contrary the exact input is transcribed without any changes. It can be questioned whether such conversions should be allowed, since there is no check for overlaps in the super structure free MS-representation. However an overlap check can easily be performed in the SC-representation before or after the conversion, which allows to generate a warning when an overlap exists or existed during a conversion. Solving overlaps during a conversion could be implemented, but this will make the conversion algorithms overcomplicated as then constraint checking is required as well. How overlaps are handled, either by user intervention or by the optimisation algorithms, is not considered yet here. It is concluded that overlaps do not cause erroneous results in the conversion algorithms and that overlaps stay in existence through the conversion.

Non-connected spaces

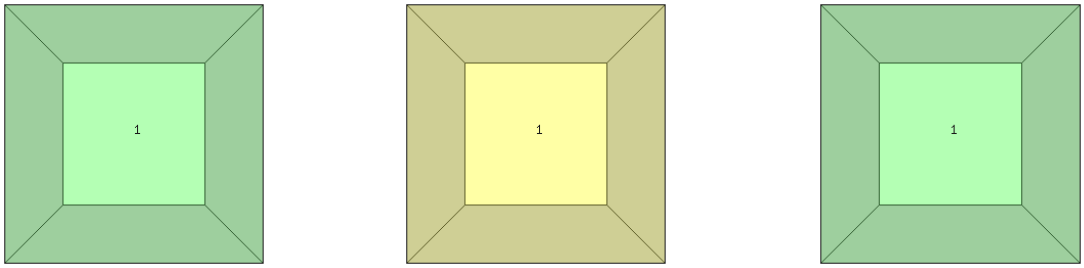
Another constraint of interest are non-connected spaces i.e. multiple building blocks in one building model, an input of two spaces is created for use in both conversion algorithms. The result is shown in Figure 2-18, it shows that non-connected spaces are converted without any changes. This will not be solved during the conversion as the algorithms would then require design modification and constraint checking, which would make the algorithms overcomplicated. It is concluded that non-connected spaces are converted without error and that the spaces stay unconnected.



2-18 Conversion of a building with separated spaces, from left to right the initial MS-representation is converted to SC-representation which is then converted back again.

Floating structures

As explained in the previous section there are no coordinates in the SC-representation, conversion to the MS-representation will find a new origin for coordinates to prevent floating structures. This is tested for one floating space that is entered as an MS-representation, the conversion results are shown in Figure 2-19. The result shows the desired result, the coordinates of the MS-space are reset after it is converted to and back from an SC-representation. Therefore any changes made to the design that would result into a floating MS-representation have no consequences for the MS-representation. It is concluded that buildings without ground connectedness will be prevented by the conversion algorithms, this does however still require the first input to be correctly entered by a user.



$S_1 \begin{Bmatrix} \{3000,3000,3000\}, \\ \{500,1000,500\} \end{Bmatrix}$

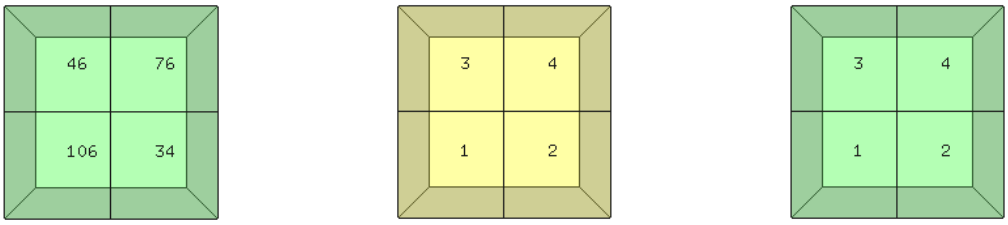
$\vec{w}\{3000\}, \vec{d}\{3000\}, \vec{h}\{3000\}$
$\overrightarrow{b^l_{i,j,k}}\{b^1_{i,j,k}\}$
$b^1_{i,j,k}\{1\}$

$S_1 \begin{Bmatrix} \{3000,3000,3000\}, \\ \{0,0,0\} \end{Bmatrix}$

2-19 Conversion of a floating space, from left to right the initial MS-representations is converted to SC-representation which is then converted back again. Each representation with their used values.

Space identification

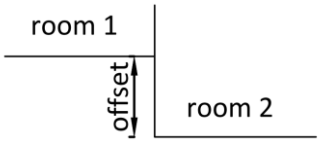
Spaces are given an identification number in the MS-representation, however in the SC-representation such identification is not required as spaces are indexed (l). After conversion the identification of an MS-space will be reset to the index number of an SC-space, this is illustrated in Figure 2-20. The loss of identification numbers is unfavourable when it is trying to track the changes of a design throughout an optimisation. It is however easily resolved by storing the space identification in the SC-representation as well, while it is not used it is saved to pass on to the following MS-representation. This is not yet implemented as it does not affect the algorithm's functionality, it is subject for future implementation when the conversions are implemented in the optimisation techniques.



2-20 Conversion of an MS-representation with arbitrary space-ID's, from left to right the initial MS-representations is converted to SC-representation which is then converted back again.

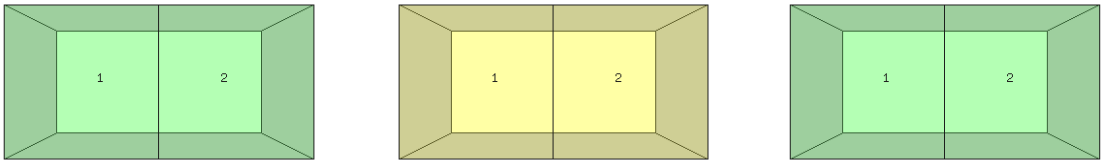
Truncation in dimension values

Dimension values may vary over a continuous domain, this means that a space may not align perfectly to another space. It is thus possible that a small offset exists in a wall that runs through multiple spaces, this is illustrated with Figure 2-21.



2-21 An offset in a wall that runs through multiple spaces

Such an offset may have a very small value, depending on the size of which the dimension values are truncated in the program. Dimension values are in the program stored by variable of type double, which can store numbers in exponential with a precision of up to 16 decimals. This is illustrated with the example in Figure 2-22, in which two spaces with a small gap in between them are converted.



$$S_1\begin{Bmatrix}\{3000,3000,3000\},\\ \{0,0,0\}\end{Bmatrix}$$
$$S_2\begin{Bmatrix}\{3000,3000,3000\},\\ \{3000.000000000002\\ ,0,0\}\end{Bmatrix}$$

$$\vec{w}\{3000, 1.819E\text{-}12, 3000\},$$
$$\vec{d}\{3000\}, \vec{h}\{3000\}$$
$$\overrightarrow{b^l_{i,j,k}}\{b^1_{i,j,k}, b^2_{i,j,k}\}$$
$$b^1_{i,j,k}\{1,0,0\}$$
$$b^2_{i,j,k}\{0,0,1\}$$

$$S_1\begin{Bmatrix}\{3000,3000,3000\},\\ \{0,0,0\}\end{Bmatrix}$$
$$S_2\begin{Bmatrix}\{3000,3000,3000\},\\ \{0,0,0\}\end{Bmatrix}$$

2-22 Example conversion of two spaces with a very small gap in between

The example shows how the minimal gap size that can be stored in the program is handled, the gap value is not stored as $2E$-12 but with a double created by arbitrary memory values during subtraction of two doubles. Besides the strange value for the gap size also the gap does not seem to exist in the MS-representation anymore after it is converted back, however this is only due to the fact that the program streams the values of doubles to file with a maximum of 2 decimals.

From the example it also becomes clear that small differences in dimension values may lead to additional cells in the SC-representation, on a larger scale this may lead to an increase of memory usage and computation time for both conversion and optimisation. It is clear that high decimal doubles can lead to strange results that may not be seen in both the visualisation and text based representation. This should be considered when using the conversions for automatic design, in which dimensions can be assigned values with any amount of decimals. A possible solution for this problem could be to disregard small values in the dimension vectors of the SC-representation (w_i, d_j & h_k). For now the problem will only be addressed, a solution will follow during implementation and validation of the conversions into the optimisation process.

2.6. *Discussion*

The topics presented in this chapter are selected for multi-disciplinary building optimisation, however argumentation for the selection of some topics have not been given. This paragraph will discuss the reasoning behind some of the selected topics. Starting with a comparison between super structure and super structure free design space representations, they both have been defined but comparisons on different levels may give a better understanding. Following, the reasons for spatial building optimisation are discussed, there are many possibilities when considering multi-disciplinary optimisation in building design. Finally the choice for the presented design space representations is discussed, which are mere suggestions for the intended research of super structured and super structure free building optimisation.

Super structured versus Super structure free design space representations

From the introduction it is clear that the difference between both representations is defined by whether the design space's size can vary or not. The varying size of a super structure free representation can be advantageous when the required number of design variables is not known a priori or when replacement of design variables should be possible. These requirements may arise from a desire to generate new and surprising designs, which is not possible when all solutions are prescribed by a super structure because all solutions would then be known beforehand. A good reason to use a super structure free design space representation is that it may lead to optima which are not devised in advance, the biggest disadvantage is though that searching the design space becomes a difficult task which cannot be handled by existing optimisation algorithms. The advantage of a super structure is that such algorithms can be used, which means that the design space can be searched effectively giving a high chance of finding the design space's optimum.

It can be concluded that in general a super structure free design space is chosen when trying to obtain new solutions that would otherwise be excluded from consideration. This in the hope that optimisation leads to new and surprising optima for the design problem. A consequence of choosing a super structure free approach instead of a super structured approach is that it requires another way of thinking about the optimisation strategies. A few aspects that are to be considered in the use of super structured and super structure free design space representations are listed below.

Implementation

Super structures are static, a programmer can use a static data structure for example matrices or arrays to describe the design space. A super structure free design space representation is dynamic, a programmer can use a dynamic data structure like a tree model with expanding branches to represent the design space.

Suitable optimisation methods

A super structured approach allows use of general methods like genetic algorithms or gradient based algorithms. A super structure free approach forces to use a few existing paradigms that are mainly used in rule based searches i.e. a heuristic search.

Memory

A super structured design space stores many or all possible solutions in memory, while many of these may not be used and thus do not contribute. In the super structure free design space it is not possible to store all possible solutions because they are not pre-described, depending on the search technique this means that only one or a few solutions are stored in memory.

Knowledge integration

The user of a super structured design approach has to prescribe al possible alternatives, which requires a considerable amount of thought. Only a set of meaningful operators needs to be define in a super structure free approach. The free approach may lead to unexpected solutions, but this also means that infeasible solutions might arise. Typically in the structured approach all solutions are prescribed as such that they are feasible.

Why spatial optimisation?

Multi-disciplinary building optimisation can be performed on many levels of design detail, some examples on an increasingly detailed level are: a spatial design for structural and building physics performance, a building frame for structural and daylighting performance, a façade for daylighting and building physics, a floor for structural and acoustics performance, a connection for structural performance and fire protection. All of the preceding examples can lead to meaningful results, however the more detailed a design optimisation becomes the less impact it has on the total building performance. This is generally recognised in the built environment, for example Hensen (2004) urges that building physics simulation software should offer users better tools for early design optimisation because even trial and error approaches are too difficult. Besides the impact of early design there is also the fact that different disciplines become less influent on each other the more the design progresses.

Multi-disciplinary optimisation for early design is however problematic as large design spaces are encountered and many disciplines influence each other. Until recently this problem has not been yet addressed because large design spaces and simulations in different disciplines quickly become expensive in terms of computer resources. Techniques and technology are however rapidly improving and the need for better building performances is increasing, therefore it is important to start research that investigates the problem. This project certainly is an endeavour to develop methods for early design optimisation, as spatial design is the earliest design decision made in a building design process and it possesses most challenges in optimisation.

Selected representations

Only two representations have been presented in this report, surely more representations could be considered. However this is not the case, the presented representations are suggestions each selected/developed for intended use of a super structure free or super structured optimisation approach. Therefore it may be possible that better design representations could be found for either the super structure free or super structured design space. For this project it is however believed that the selected representations are sufficiently adequate for their intended use, also because they can successfully be mapped to each other. Other representations may be suggested by the critical reader, it is important for future research that other representations are considered as well. Here the presented representations are used to investigate multi-disciplinary spatial building optimisation and design space, they are not particularly subject to research themselves.

Application

At the time of writing, the conversion algorithm has already been applied in the automated evaluation of building designs. Blom et al (2016) use it in the optimisation of a building layout that is represented by a super cube.

3. Building physics analysis

Extending building optimisation with building physics (BP) optimisation requires a program that can handle the BP-analysis of the building model at hand. Therefore a consideration is made in this chapter for the simulation method that is to be implemented in the optimisation toolbox. From this consideration it is concluded that a simple method is the best choice for the toolbox.

A state space representation for the thermal network of a building is selected and implemented in a C++-program that is developed for the optimisation toolbox. The working of the program is also discussed in this chapter and accordingly it is verified by means of an existing building physics simulation program.

3.1. *Building physics simulation*

In this section it is tried to gain an understanding of building physics simulations, e.g. what are they used for and how do they simulate different aspects of a building. Accordingly it is discussed which scope of these simulations should apply to the optimisation toolbox. Finally some findings of a preliminary study on different building physics tool are presented.

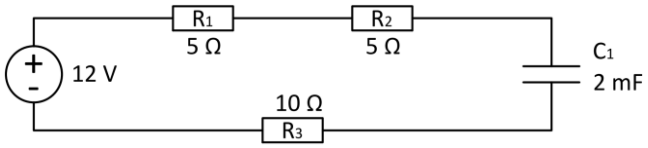
Envelope of building physics simulations

Building physics (BP) is a wide field of research that, in a nut shell, covers heat, air and moisture in a building, a BP-simulation computes aspects of one or more of those subjects. Calculations may vary from simple heat resistance calculations of a wall to complete heating, ventilation and air-conditioning (HVAC) simulations of a building, including the smallest of details. Typical calculations in a BP-analysis are listed below, in these calculations a distinction could be made between static and dynamic analysis.

Calculation	Static	Dynamic
Heat transfer	Temperature curve in a construction at specific ambient temperatures	Temperature development in a construction over time with varying ambient temperatures
Mass transfer (Ventilation)	Summation of all gains and losses of air content over a normalized time step	Gains and losses of air temperature and contents are simulated over time
Moisture	Condensation in or on a construction at specific ambient temperatures	Condensation in or on a construction over time with varying ambient temperatures and air properties
Solar heating/ irradiation	-	Temperature gain by solar radiation on a construction over time
Daylight	Sufficiency of daylight or exceedance of glare at specific times or with normalized values	Sufficiency of daylight or exceedance of glare simulated over time

These calculations may be performed independent from each other, although moisture calculations are strongly dependant of heating and ventilation analysis. BP-analysis becomes stronger when calculations are combined, as each calculation has some influence on the other. For example when a heating, ventilation and moisture analysis are combined it can be determined how much ventilation is needed in a space with cold walls to prevent condensation problems in the space.

Linking the analyses of spaces or zones makes it possible to analyse an entire building, therewith a large linked analysis is created that performs all sub analyses over small time steps. The environment of zones and constructions are then either linked to the analyses of a foregoing time step or to a predefined profile of e.g. temperature. Normally a resistor and capacitor network (RC-network), see figure 3-1, is used to map relations between different environments. An RC-network is a web of capacitors - *in the case of a building: zones, walls, floors etc*. – and resistors – *all what separates capacitors e.g. air and constructions* –.



3-1 Example of an RC-network, known from representations of electric circuits

Another approach could be the use of machine learning techniques like neural networks. This approach lets an algorithm determine a system that can simulate the thermal behaviour of a building. However, machine learning requires (real world) data, which serves as learning material for the algorithm.

BP-simulations create a time profile of specific properties of a building with which predictions about the performance of building energy usage, comfort and safety can be determined. These predictions are made by an abstraction of the real building, e.g. an RC-network. The accuracy of a prediction is determined by the refinement and detail of the abstraction, for example in how many layers should a wall be divided or how many layers in the case of neural networks. A detailed model or input of a building is required when accuracy

is desired, vice versa, when accuracy is less important then a more superficial building model can be used. It generally applies that undetailed building models do not benefit from extensive simulation.

Scope

As stated, the scope of a BP-analysis strongly depends on the level of detail in the building model. In this research a building is represented only by spaces that are assigned simple properties like a wall, floor or ceilings by a rule based design. Therefore the model cannot be considered detailed and an extensive analysis would be superfluous, but now the question remains: to what extent should the analysis be performed?

The data required from the building model should first be considered for each type of analysis in an RC-network analysis. Table 1 lists the considerations made in the choice of what type of simulation method should be used in the project.

Table 1 Considerations in the choice of a simulation method

Type of analysis	Required data	Considerations
Heat transfer	Construction details	Construction details like material and dimensions are, up to some extent, already in the model.
	Outside temperature	Meteorological data is available, a location has to be added to the model, which is plausible as a location is a mandatory choice in the design process, if not a boundary condition.
	Inside temperature	There are 2 options: - Use a heating plan, coupled to a user profile - Use a temperature profile A heating plan based on a user profile, requires space functions and user data. A temperature profile as function of time may already give a good estimate
	Heating/cooling plant	A system that controls heating and cooling of the building should be in place, this can be as easy as a virtual power source at a capacitance to more complex heating systems like a central unit providing the demand throughout the building
	Solar irradiation	Meteorological data is available, also see outside temperature
Ventilation	Zone/space details	Zone details like dimensions are already in the model
	Inside/outside temperature	See heat transfer above
	Ventilation demand	There are two options: - Use a ventilation plan, coupled to a user profile - Use a constant ventilation profile A heating plan based on a user profile, requires space functions and user data.
	Ventilation installation	A system providing the ventilation demand should be in place, this can be as easy as a mass flow in air changes per hour to the simulation of a ventilation unit with heat recovery and moisturiser
Moisture	Construction and zone details	Details like dimensions or materials are, up to some extent, already in the model
	Moisture content of air	This depends on occupancy, furniture, air temperature and more. All together this depends on heating and ventilation of a building and requires accurate calculations of heating and ventilation
Daylight	Solar irradiation	Meteorological data is available, also see outside temperature

	Construction topology	The topology of the construction, e.g. layout of the façade and location of windows may be too specific, as relationships between different building elements should be made. However the information is present, it must be interpreted first.
	Shading information	Parts of the building that create shadows on other parts require some sort of ray tracing, which is a relatively extensive computational process.
General	User profile	A user profile requires information about the function of a building, e.g. it an office or a residential building. This information is considered too specific for the building model at hand

From the preceding table it can be concluded that a simulation of heat transfer and ventilation are possible with the building model at hand. A simulation of moisture requires high accuracy in the heat and ventilation simulations, thus a moisture simulation will not be included. A daylight calculation is possible, however it is complicated by shadow effects induced by the building and thus is less suitable. A user profile could be added to the building model, however the level of detail required for such a profile is not proportional to the level of detail in the building model at hand. It is decided that a simulation will only include heat transfer and ventilation, data that needs to be added to the building model are:

- Meteorological data: Data regarding date, time, outside temperature and solar radiation are stored in files. This file also indirectly determines a location, namely the location at which the measurements were made.
- Heating and cooling plant: Least detailed method is an idealised source in each space, data like temperature set points and heating/cooling power are required.
- Ventilation: Least detailed is an infiltration rate in air changes per hour, only data required per space is the number of air changes per hour.
- Material and construction properties: Although construction types are already in the building model, these types do not yet hold properties related to the building physics analysis.
- Simulation period: A start and finish date for the simulation period of which meteorological data is used.

The type of simulation has now been determined, only a selection of a program remains. In a preliminary study several programs have been tested for their compatibility with the optimisation toolbox. This preliminary study and its conclusion are shortly discussed hereafter.

Program selection, preliminary study

In the preliminary study, three programs were tested: ESP-r, Energy Plus and HAMBASE. A few simulations were carried out to test whether programs are suitable for a building model in the "Movable Sizable" representation. Subsequently, the programs were reviewed on the following topics:

- Platform compatibility, what OS is required, e.g. Windows or a UNIX platform? Preferably the program works cross platform.
- Program licensing, can the program be used freely for scientific research?
- Program integration, can the program be integrated in the toolbox?
- Required data, what is the minimal required amount of data to carry out a simulation? The more data is needed, the mode data needs to be added to the building model

All programs considered it was concluded that a fourth option would suit the toolbox best, namely a self-written simulation program. The considerations per program that led to this conclusion are discussed below, i.e. the reasons why a program was not suitable or a self-written program would be better suited.

ESP-r

This program is UNIX based, it can be run on Windows by either a native windows installation or by running a UNIX simulator like Cygwin on Windows. A cross platform integration into the optimisation toolbox is with two differently compiled programs quite cumbersome. From the test simulations it was also concluded that creating input for a simple model is elaborate, there are many input parameters of which it is not always clear whether they should be used or not. In addition, it should be noted that it was not succeeded to match the ESP-r results with those of the other two programs, which themselves did show similar results. Integration into the toolbox would take the form of the creation of an input file for the ESP-r program that is then fed to the program, accordingly an output file must then be read by the toolbox. A free licence for scientific use is available, however this may have consequences for the use and disclosure of the toolbox. Finally, ESP-r may be subject to updates, this could have consequences for the continuity of the version used in the toolbox.

It is concluded that ESP-r can be incorporated in the toolbox, however developing a cross platform incorporation is expected to be too elaborate. Subsequently, ESP-r input has shown to be complicated and incomprehensive. It is unclear of what data is needed for the input of a simple model and it seems that, in comparison, too much data must be added to the building model of the toolbox. Integration by means of input and output files is also not optimal when regarding the fact that optimisation may require many separate simulations.

Energy Plus

This program is developed for Windows, but a version for Linux is also available. Again two different compilations of a program may cause problems when integrated in a cross platform toolbox. The input of the program is clear, however there are still a lot of parameters to be accounted for in the input. The simulation results showed good comparison with those of HAMBASE in the case of a simple building model. Integration into the toolbox is possible by generation of an input file, that then must be fed to the simulation program, subsequently and output file must be read by the toolbox. Energy plus has a free licence for scientific use, however this may have consequences for the use and disclosure of the toolbox. Finally, Energy Plus may be subject to updates, which could have consequences for the continuity of the version used in the toolbox.

It is concluded that Energy Plus may very well become a part of the toolbox, although the development of a cross platform integration can prove troublesome. The input of Energy Plus is comprehensive, however there are many parameters to account for in the input, which could mean a lot of additional data is needed in the building model. Integration by means of input and output files is also not optimal when regarding the fact that optimisation may require many separate simulations.

HAMBASE

This program is based on MATLAB code, MATLAB has versions for both windows and UNIX. Integration via MATLAB could be done by passing arguments to the OS to run a MATLAB code. This would mean that an input file in the form of a script and an output file in text are required. Another option is to translate the MATLAB code into the code in which the toolbox has been written, namely C++. Translation into comprehensible code is however a laborious process, even though an automated C++-code generator is available in MATLAB. HAMBASE does not need an elaborate input some excessive functions and parameters for the purpose at hand exist, but these do not complicate the generation of input. HAMBASE is a scientific work, and thus consequences for use and disclosure of the toolbox may be less than ESP-r and Energy Plus. However HAMBASE has a dependency on MATLAB, which has a commercial license, this will lead to an impairment on the use of the toolbox. Although updates may be applied to the code, this does not have consequences for the toolbox. Since the code itself will become part of the toolbox, there is no dependency on the availability of older versions of the code when installing the toolbox.

It is concluded that HAMBASE is best suited for the building model at hand, a simple and comprehensive input is asked. However the program is not suitable for an integration into the toolbox, since either a license for MATLAB or a laborious translation of the code are required.

A self-written code comes into consideration when regarding the fact that integration of any of the considered programs into the toolbox is elaborate. The main advantage of a self-written code is that it is designed solely for its task within the toolbox, it is much more compatible with the model as it is and output can be extracted as desired. Other advantages are: not sensitive to updates, can be platform independent, no licences, full control over simulation algorithm and complete insight in the program's functions and parameters.

Simulation method

With the choice of writing a simulation program comes the choice in simulation methods. A focus has been placed on simplified simulation methods. Kramer et al. (2012) present a literature review on simplified thermal and hygric building modelling. They make a distinction between three types of simulation models: Neural networks, linear parametric and a resistor capacitor (RC) network models. The first two require real world data in order to define the simulation model, the model in turn does not have any physical meaning. An RC-network is built by elements of physical meaning and thus does not need real world data, this makes an RC-network simulation model suitable for the toolbox. How such a network is used to simulate a building is elaborated in the next section.

3.2. *State space approach on a thermal RC-network of a building*

System of ordinary differential equations

As stated, most building physics simulations use an RC-network to represent the thermal properties of a building. The state of each element in an RC-network can be expressed by an ordinary differential equation, the network together thus forms a system of ordinary differential equations.

State space representation of a dynamic problem

A state space representation is a model of a time dependant system that can be described by differential equations e.g. electrical circuits, physical problems, mechanical problems or heat flow problems. The state space method discretises time, which is done in the solver, the time variable can therefore often be omitted in the representation itself. Equations (13) and (14) below are the set of equations that is characteristic for the state space representation.

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\cdot\boldsymbol{x}(t) + \boldsymbol{B}\cdot\boldsymbol{u}(t) \qquad OR \qquad \dot{\boldsymbol{x}} = \boldsymbol{A}\cdot\boldsymbol{x} + \boldsymbol{B}\cdot\boldsymbol{u} \tag{13}$$

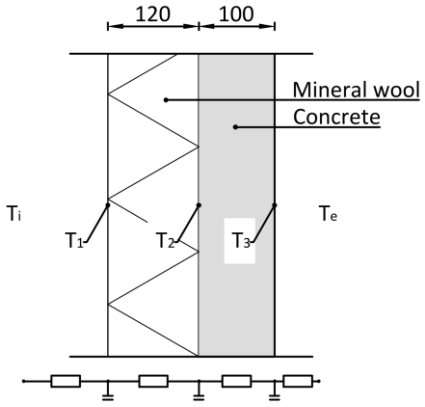
$$\dot{\boldsymbol{y}}(t) = \boldsymbol{C}\cdot\boldsymbol{x}(t) + \boldsymbol{D}\cdot\boldsymbol{u}(t) \qquad\qquad \dot{\boldsymbol{y}} = \boldsymbol{C}\cdot\boldsymbol{x} + \boldsymbol{D}\cdot\boldsymbol{u} \tag{14}$$

The first equation describes the states of a system e.g. voltage, velocity, acceleration or temperature. In higher order differential equations states can depend on each other, for example velocity and acceleration can both be a state where the velocity is dependant of the acceleration. The second equation describes the output of the system i.e. each state that is of interest for the user, this way the represented system is completely independent from the requested output.

Using the state space method comes down to determining the $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ matrices, this is done by first determining all states ($\dot{x}_1$ to $\dot{x}_n$) in the system including the input (u_1 to u_m), followed by the output ($\dot{y}_1$ to $\dot{y}_p$). $\mathbf{A}$ is then an n by n matrix that describes the system dependency on the states, $\mathbf{B}$ is then an m by n matrix that describes the system dependency on the input. $\mathbf{C}$ is a p by n matrix describing what states are requested as output and $\mathbf{D}$ is a p by m matrix describing which input values are requested as output.

Worked example of a single wall

A small example is solved in this section, the state space method is used to simulate warming up of a construction ($5℃$) after it is suddenly exposed to a constant internal temperature ($20℃$) and a constant external temperature ($-10℃$). The construction along with its properties are shown in Figure 3-2 below, the thicknesses shown are in millimetres, the calculations will be done per square meter of wall surface.

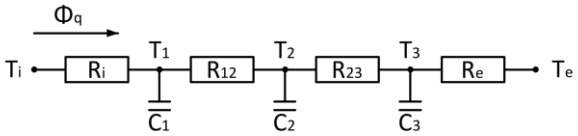


	Mineral wool	Concrete
Thermal conductivity *λ - [W/(K m)]*	0.04	1.8
Specific weight *ρ - [kg/m³]*	20	2400
Specific heat *C - [J/(K kg)]*	2000	850

3-2 Arbitrary construction of an exterior wall with its thermal properties

The temperatures at the construction during the warm up period are of interest for this problem, i.e. T_1, T_2 and T_3. The construction is discretised into layers to be able to compute the temperature over the construction thickness, the construction layers can conveniently be used for this discretisation. The

discretised model of the example can be represented by a network of resistors and capacitors, as is done with electrical circuits in the field of electrical engineering, see Figure 3-3.This problem is not represented by a closed circuit, however the temperature difference over time between outside and inside is known a priori. An expression for the heat flux Φ_q can thus be composed using the resistances and temperature differences.



3-3 RC-network of the construction

The values of the resistances can be calculated per discrete layer, the capacitance at each point of interest is a lumped capacitance of the surrounding construction. The lumped capacitance in this example will be computed proportional to the distances between the measurement points, i.e. from midpoint to midpoint of each adjacent layer.

$$C_1 = \left(\frac{1}{2}\cdot 0.120\right)\cdot 2000 \cdot 20 = 2400\ J/K$$

$$C_2 = \left(\frac{1}{2}\cdot 0.120\right)\cdot 2000 \cdot 20 + \left(\frac{1}{2}\cdot 0.100\right)\cdot 850 \cdot 2400 = 162.5 = 104400\ J/K$$

$$C_3 = \left(\frac{1}{2}\cdot 0.100\right)\cdot 850 \cdot 2400 = 42.5 = 102000\ J/K$$

$$R_i = 0.13 = 0.13\ K/W$$

$$R_{12} = 0.120/0.04 = 3.00\ K/W$$

$$R_{23} = 0.100/1.8 = 0.056\ K/W$$

$$R_i = 0.04 = 0.04\ K/W$$

Only the state space system remains to be put together, starting with the states at the capacitors. Equating the flux difference at each capacitor to the difference in flux between the two adjacent resistances at each measurement point results in Equations (15) to (17).

$$C_1 \cdot \frac{dT_1}{dt} = \frac{T_i - T_1}{R_i} - \frac{T_1 - T_2}{R_{12}} \quad (15)$$

$$C_2 \cdot \frac{dT_2}{dt} = \frac{T_1 - T_2}{R_{12}} - \frac{T_2 - T_3}{R_{23}} \quad (16)$$

$$C_3 \cdot \frac{dT_3}{dt} = \frac{T_2 - T_3}{R_{23}} - \frac{T_3 - T_e}{R_e} \quad (17)$$

These equations hold all 3 states in the state space representation, namely the temperatures at each capacitor. Thus the above equations can be rewritten to the desired representation of the state space method, resulting in equations (18) to (20).

$$\dot{x}_1 = \frac{dT_1}{dt} = \frac{T_i}{C_1 \cdot R_i} - \frac{T_1}{C_1 \cdot R_i} - \frac{T_1}{C_1 \cdot R_{12}} + \frac{T_2}{C_1 \cdot R_{12}} \tag{18}$$

$$\dot{x}_2 = \frac{dT_2}{dt} = \frac{T_1}{C_2 \cdot R_{12}} - \frac{T_2}{C_2 \cdot R_{12}} - \frac{T_2}{C_2 \cdot R_{23}} + \frac{T_3}{C_2 \cdot R_{23}} \tag{19}$$

$$\dot{x}_3 = \frac{dT_3}{dt} = \frac{T_2}{C_3 \cdot R_{23}} - \frac{T_3}{C_3 \cdot R_{23}} - \frac{T_3}{C_3 \cdot R_e} + \frac{T_e}{C_3 \cdot R_e} \tag{20}$$

In these equations T_i and T_e do not belong to a state and as mentioned early are known a priori, thus they belong to the input vector $\mathbf{u}$. Writing the state equations above into matrix format results in Equations (21) and (22), which yields the $\mathbf{A}$ and $\mathbf{B}$ matrices that are required as input for the state space solver.

$$\mathbf{A} \cdot \mathbf{x} = \begin{bmatrix} \frac{-1}{R_i \cdot C_1} - \frac{1}{R_{12} \cdot C_1} & \frac{1}{R_{12} \cdot C_1} & 0 \\ \frac{1}{R_{12} \cdot C_2} & \frac{-1}{R_{12} \cdot C_2} - \frac{1}{R_{23} \cdot C_2} & \frac{1}{R_{23} \cdot C_2} \\ 0 & \frac{1}{R_{23} \cdot C_3} & \frac{-1}{R_{23} \cdot C_3} - \frac{1}{R_e \cdot C_3} \end{bmatrix} \cdot \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} \tag{21}$$

$$\mathbf{B} \cdot \mathbf{u} = \begin{bmatrix} \frac{1}{R_i \cdot C_1} & 0 \\ 0 & 0 \\ 0 & \frac{1}{R_e \cdot C_3} \end{bmatrix} \cdot \begin{bmatrix} 20 \\ -10 \end{bmatrix} \tag{22}$$

The dynamic system has successfully been described by the equations above, any desired output regarding the states or input can now be defined. In the problem statement at the beginning of this section it is declared that T_1, T_2 and T_3 are of interest, resulting in equations (23) to (25).

$$\dot{y}_1 = T_1 \tag{23}$$

$$\dot{y}_2 = T_2 \tag{24}$$

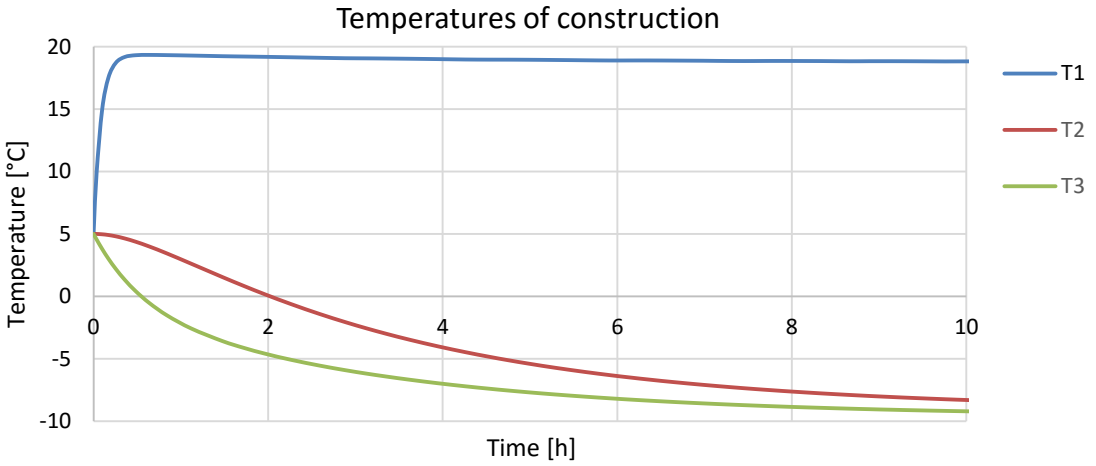
$$\dot{y}_3 = T_3 \tag{25}$$

Writing the above in matrix format results in Equations (26) and (27), these yield the $\mathbf{C}$ and $\mathbf{D}$ matrices that are required as input for the state space solver

$$\mathbf{C}\cdot\mathbf{x}=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\cdot\begin{bmatrix}T_1\\T_2\\T_3\end{bmatrix} \tag{26}$$

$$\mathbf{D}\cdot\mathbf{u}=\begin{bmatrix}0&0\\0&0\\0&0\end{bmatrix}\cdot\begin{bmatrix}20\\-10\end{bmatrix} \tag{27}$$

All ingredients to simulate the system are now available, the derived equations and properties are used in a MATLAB script that calls a state space solver in Simulink accordingly, see Annex 5. For each measurement point a graph is plotted over time, see Figure 3-4. The graphs show a quick warm up time for the inside surface temperature, which is due to the low capacitance of the insulation material. The concrete layers take much longer to cool down despite the low heat resistance, which is due to the high capacitance.



3-4 Simulation results from MATLAB, simulated warming up of the construction

The above simulation can be checked by calculating each temperature after the warm up time, this can be done manually since at that time there is no heat flux remaining in the construction. The temperature increment in each layer is calculated proportional to the resistance of that layer, thus the temperature increment at layer j can be calculated with expression (28).

$$\Delta T_j = \frac{R_j}{\sum_{j=i}^{n} R_n} * (T_e - T_i) \quad with\ n \in \{i, 1, 2, 3, e\} \tag{28}$$

Doing so for each layer results in the values given in Table 2, which tabulates per layer the temperature at the start of that layer - moving from inside to outside - next to temperatures simulated with the state space approach.

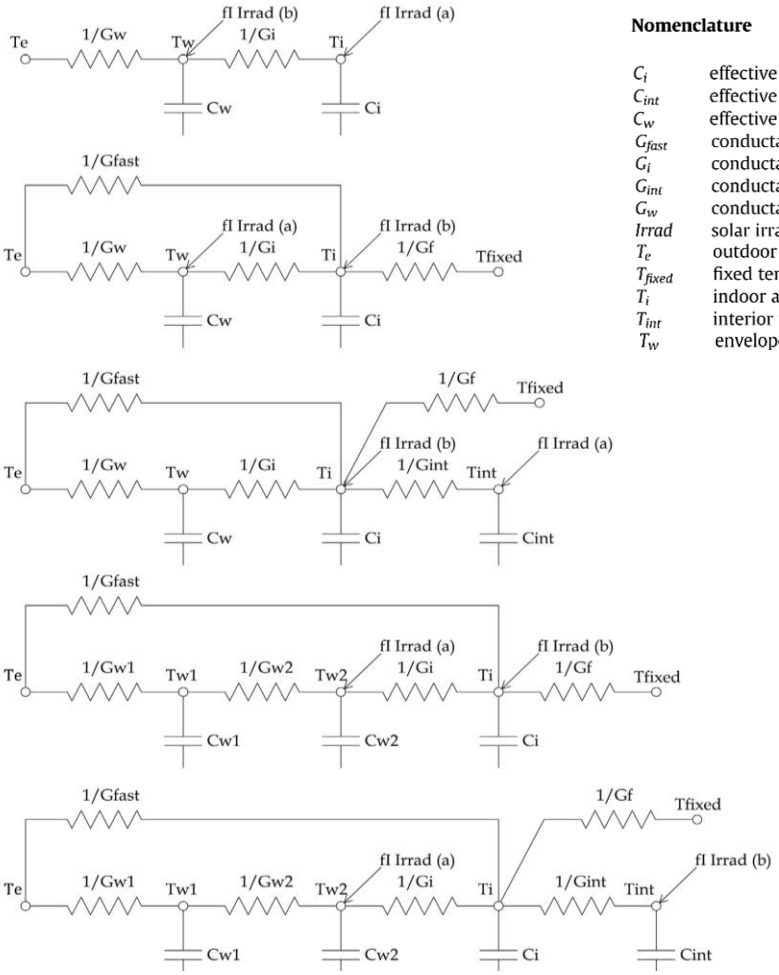
Table 2 Temperatures at each measurement point after the construction is fully warmed up

Layer [-]	Resistance $[K/W]$	Temperature increment [°C]	Temperature (t = ∞) [°C]	Simulated (t = 24 h) [°C]
R_i	0.13	-1.21	20	-
Mineral wool	3.00	-27.90	18.79	18.79
Concrete	0.056	-0.52	-9.11	-9.10
R_e	0.04	-0.37	-9.63	-9.62
total:	3.226			

The results obtained from the hand calculation correspond with the simulated temperatures, this does however not validate the simulation model. As the capacitances of each layer are discretised there may be a delay or increase in warm up time, it is not checked here but the effects may be reduced by mesh refinement.

RC-network of a building

The preceding example clarifies how an RC-network can be used to simulate the temperature state of a wall. A wall is a link of different layers of that wall, connecting an outside temperature states with an inside temperature state. A building is in the same way a system of links between different temperature states. As such all the different components that create a temperature state should be identified. A distinction between two types of states can be made i.e. space/air temperature and construction temperature. A general RC-building block can be made with this distinction, in which two air or space state are separated by one construction state, see figure 3-5.

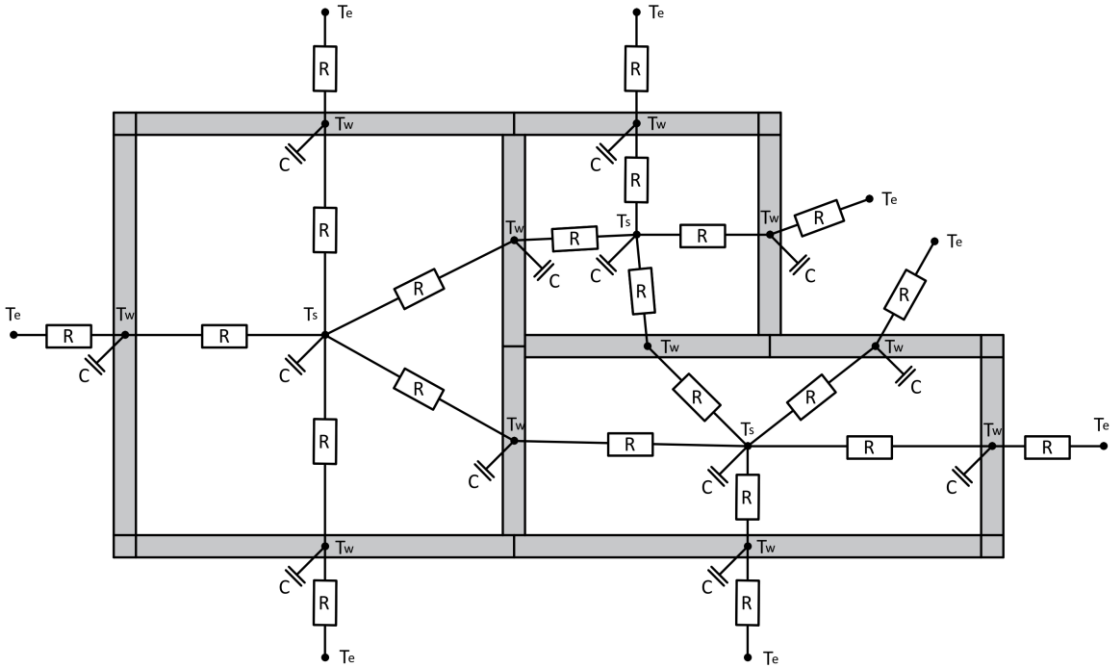


Nomenclature

C_i	effective indoor air capacitance [J/K]
C_{int}	effective interior capacitance [J/K]
C_w	effective envelope capacitance [J/K]
G_{fast}	conductance from indoor air to outdoor air [W/K]
G_i	conductance from indoor air to envelope [W/K]
G_{int}	conductance from indoor air to interior [W/K]
G_w	conductance from envelope to outdoor air [W/K]
$Irrad$	solar irradiation [W/m^2]
T_e	outdoor air temperature [°C]
T_{fixed}	fixed temperature [°C]
T_i	indoor air temperature [°C]
T_{int}	interior temperature [°C]
T_w	envelope temperature [°C]

3-5 RC-building blocks, source: Kramer et al. (2013)

A complete RC-network is then built by a concatenation of such building blocks, as an example the first building block from the preceding figure is used to compose an RC-network of a simple floorplan, figure 3-6. No solar irradiation or heating/cooling irradiation is depicted in the example, however a distinction between walls with different orientation is made for purposes of solar irradiation.



3-6 RC-network of an arbitrary building layout

From the example it becomes clear that each space has a capacitance, i.e. the air and furniture inside the space. Each wall has only one capacitance, i.e. all material layers of that wall. The resistances between the states consist of the resistance by a construction plus an air (transition) resistance. The resistance between a wall and another state depends on where the temperature of the wall is simulated, usually this point is midway the thickness of a wall.

Now a general idea of a building physics simulation method has been outlined. The next section discusses the developed BP-simulation program that composes and simulates an RC-network by means of a state space approach.

3.3. *State space representation of an RC-network in C++*

C++ is a programming language for which open source libraries exist. Solutions to programming problems are often realised by such libraries. ODE solvers are no exception to open source libraries, here the odeint solver - part of the Boost libraries - is selected for the problem at hand. Boost is a collection of peer reviewed libraries that are intended for a wide variety of uses in many types of applications. The Odeint solver is developed to be generic by template metaprogramming, which makes it applicable to nonstandard problems and by which integration with other data structures and libraries are allowed. Odeint is a numerical solver in which multiple solver algorithms can be selected for example the Runge Kutta, Rosenbrock or Adams algorithms are incorporated for solving explicit, implicit or symplectic problems.

The building physics analysis program is written by use of object oriented programming. As such temperature states are treated as objects, the ODE-system is therefore composed of multiple states/objects. The ODE-solver is called to solve the composed system. This section treats the features that are currently supported by the program, and what features are expected to be implemented in the future. Accordingly the structure of the program is discussed, first the main structure of the program is clarified and then the composition and solving of the system are clarified.

Features

The developed program's features are a thermal simulation, a weather profile, constant ventilation and an idealised heating/cooling system. The RC-network is constructed by the second building block in figure 3-5, with absence of solar irradiation and addition of heating or cooling radiation at each T_i.

Input for the program is adduced by an input file, figure 3-7, in which information about the simulation period, exterior temperatures, materials, constructions and the states are stored. Some of this information, e.g. simulation period, materials or construction must be defined by a user. However the information about the states can be obtained from the building model at hand. How this information is obtained will not be discussed until chapter 4.

```
#Warm up days
U,0

#Time steps per hour
T,4

#Weather,       Start year,     Start month,    Start day,      Start day name, Start hour,     Finish year,    Finish month,   Finish day,     Finish hour
E,              1985,           9,              1,              default,        1,              1985,           10,             1,              24

#Ground profile,Temperature
G,              50

#Materials,     Material_ID,    Name,           Spec_Weight,    Spec_Heat,      Thermal_Conduc
M,              1,              Concrete,       2400.0,         850.0,          1.8
M,              2,              Isolatie,       60.0,           850.0,          0.04

#Constructions, Construct_ID,   mat_ID_1,       thickness_1,    ...........     mat_ID_n,       thickness_n
C,              1,              1,              100.0,                          2,              50.0

#Spaces,        Space_ID,       Volume [m³],    Heating [W/m³], Cooling [W/m³]  Heat set point  Cool set point  Air changes/hour
S,              S_1,            27,             0.0,            0.0,            20.0,           22.0,           0.0

#Walls,         Wall_ID         Construction    Area [m²],      Orientation,    Space side one, Space side two
W,              W_1,            1,              9.0,            0,              S_1,            G
W,              W_2,            1,              9.0,            90,             S_1,            G
W,              W_3,            1,              9.0,            180,            S_1,            G
W,              W_4,            1,              9.0,            270,            S_1,            G


#Windows,       Window_ID,      Area [m²],      U_value [K/m²W] Cap./Area,      Orientation     Space side one, Space side two

#Floors,        Floor_ID,       Construction,   Area,           Space side one, Space side two
F,              F_1,            1,              9.0,            S_1,            G
F,              F_2,            1,              9.0,            S_1,            G
```

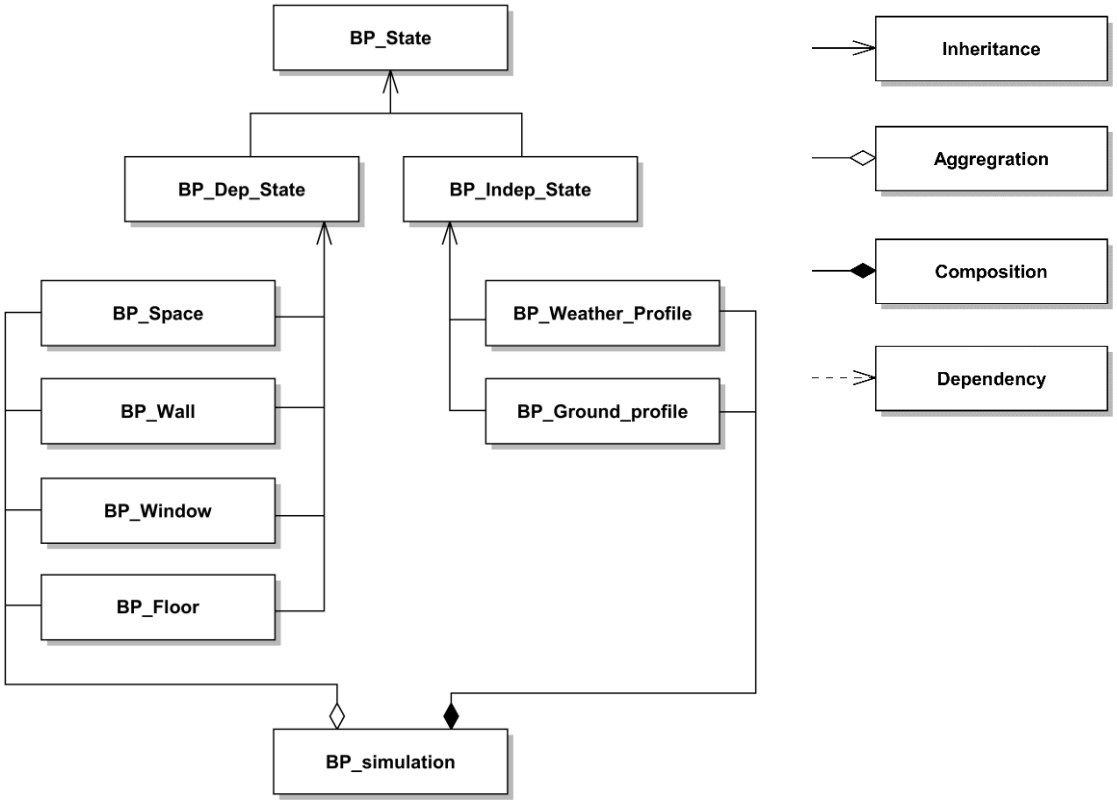
3-7 Example of the input file of the simulation program

Features that are not included but are considered for implementation are: solar irradiation and a date/ time tracker to enable possible profiles for heating, cooling or ventilation. These are not yet implemented because they are not mandatory for optimisation purposes and an implementation of the program into the toolbox is prioritised.

Main structure of the program

The main structure of the program is depicted in figure 3-8. The figure shows how different temperature states are inherited from their base classes, and how they eventually compose the system *(BP_Simulation*). Inheritance here allows the use of polymorphism, which means that an inherited class can be called upon by its base class. This is exploited during the composition of the state space matrices and additionally it allows

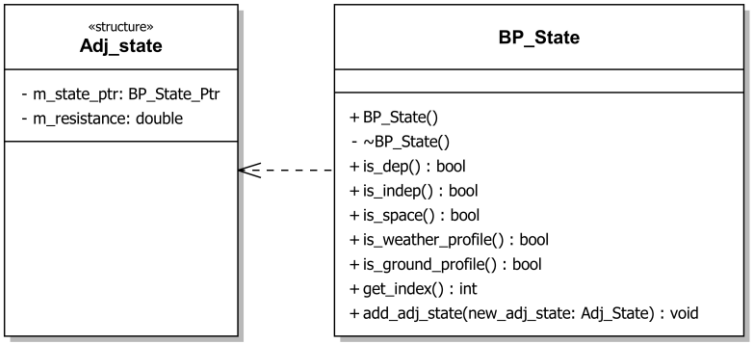
the system to be extended with new types of temperature states. The structure in the figure above also shows that a thermal RC-system here is an aggregation of multiple dependant states, e.g. walls and spaces, and a composition of a specific number of independent states, e.g. outside temperature and the ground temperature. The simulation is eventually carried out by functions of the class that contains the state classes, which consists of updating the state matrices with state independent data and solving the system subsequently. The following headings will discuss each class in a more detailed manner, which will clarify the workings of the program. The actual code of the program has been added to Annex 4



3-8 Main layout of the heat building physics analysis program

BP_State

The first base class is a pure virtual class, meaning its only function is to be a base class to other (derived) classes. The purpose of this class is therefore to give the derived classes the desired polymorphism properties. The figure below, figure 3-9, shows the class diagram of the BP_state class.

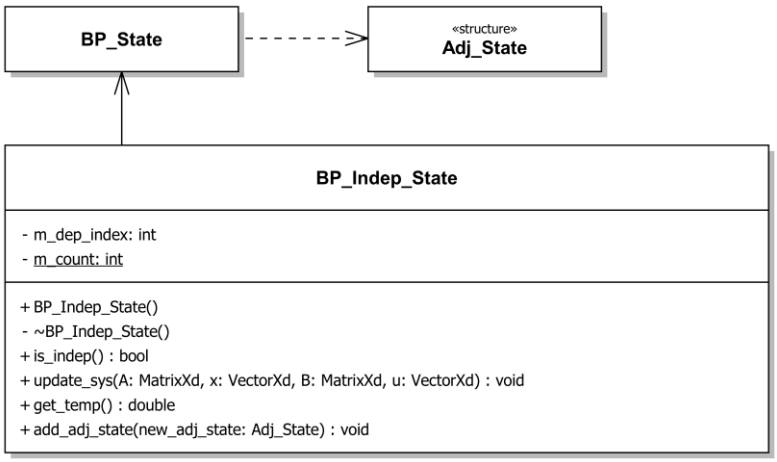


3-9 Class diagram of the *BP_State* class

From the diagram it becomes clear that the main function of the *BP_State* class is the identification of an object with base class *BP_state*. Besides identification this class is also used to make the function which maps relationships with other states virtual. This class does not own any member variables, the only variable they have in common is temperature. The temperature variable is however held in a matrix in the *BP_simulation* class, because the ODE-solver cannot handle dynamically allocated variables in the ODE-function.

BP_Indep_State

The independent state class is a pure virtual class as well, the class's purpose is to represent states that belong to the input vector $\mathbf{u}$ of the state space representation.

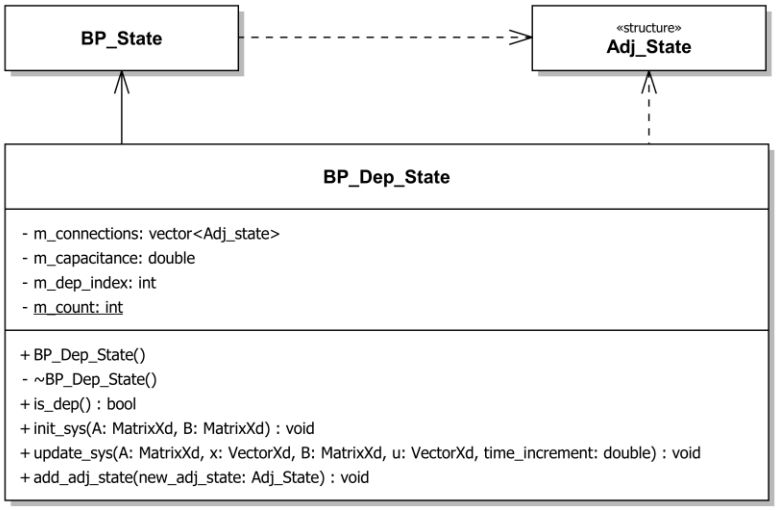


3-10 Class diagram of the *BP_Indep_State* class

Figure 3-10 shows the introduction of member variables, which are intended to index instances of the class. This index then corresponds to a temperature value in the input vector $\mathbf{u}$. Some functions from the base class are overloaded, however a new function is declared as well. The update_sys() function updates values in the state matrices and vectors where appropriate.

BP_Dep_State

The dependant state class is also a pure virtual class, its purpose is to hold states that belong to the state vector $\mathbf{x}$ of the state space representation.



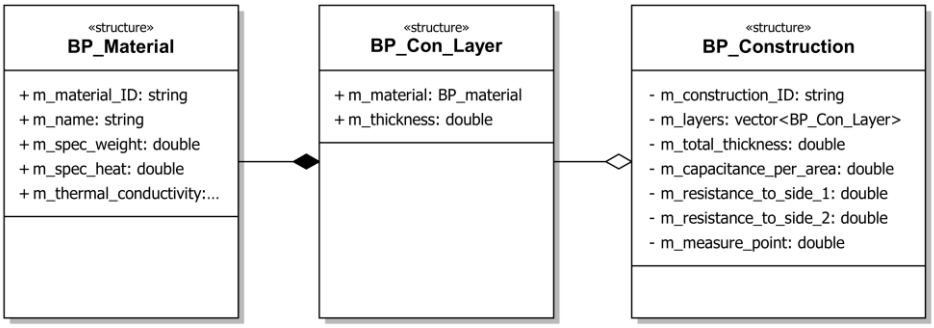
3-11 Class diagram of the *BP_Dep_state* class

Figure 3-11 again shows member variables that provide the indexing of class instances, now representing state vector $\mathbf{x}$. However, variables concerning resistances and capacitances are introduced, which compose the basis of the state matrices $\mathbf{A}$ and $\mathbf{B}$. In the class diagram the capacitance of a state is given by a single variable, but a vector stores information about adjacent states with the appropriate resistance between states (also see figure 3-9).

Again, some functions from the base class are overloaded, and here also the *update_sys()* function is declared. Another function is declared here as well, *init_sys()*, this initialises the A and B matrices with the values of the resistances and capacitance that are assigned to an object of the class at the appropriate indices.

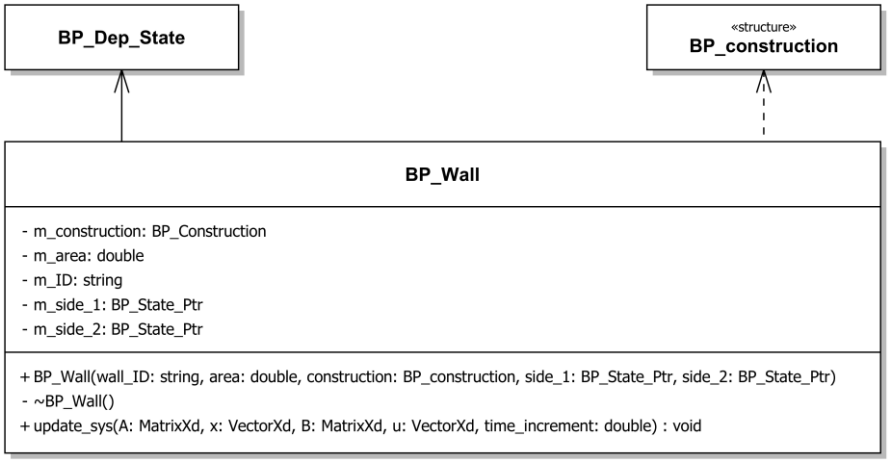
BP_Wall

The wall class is inherited from *BP_Dep*, the class is not virtual, it represents a wall with all the properties required for a thermal simulation. The defining property of a wall is its construction, for this a special system of structures is declared.



3-12 Class diagram of the *BP_Construction* structure and its associated structures

A construction is defined by layers of materials, as such a construction is composition of multiple layer structures that each are an aggregation of a thickness and the *BP_Material* structure. The material structure holds all properties of influence to the thermal simulation. On initialisation of an object of the construction structure, layers are assigned, accordingly variables like thickness, capacity and resistances can be determined.



3-13 Class diagram of the *BP_wall* class

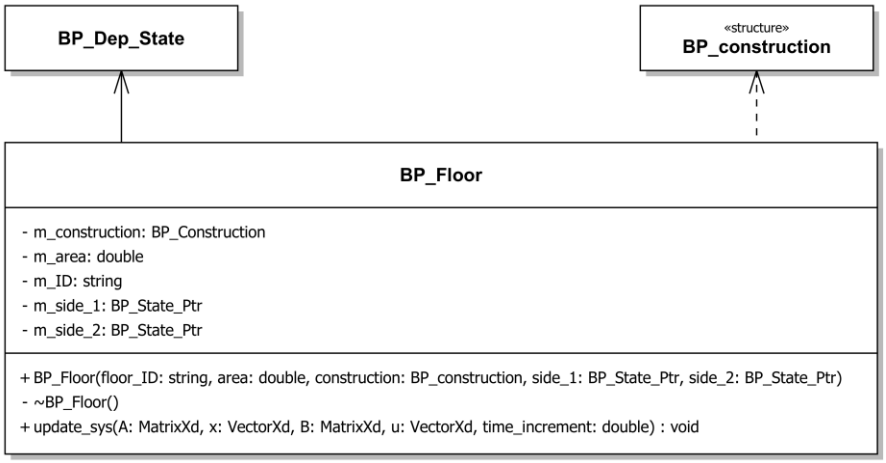
The member variables of the wall class are what is needed to identify the specific wall and to determine the values for the base class: *BP_Dep_State*. As such, the heat capacity of a wall is easily calculated, equation (29), by multiplying the capacitance per area from the *BP_Construction* structure with the wall's area.

$$C = C_{per\ area} \cdot A \tag{29}$$

In the above expression C is the capacitance in $[J/K]$, $C_{per\ area}$ is the capacitance per area in $[J/(K \cdot m^2)]$ and A the wall's surface area in $[m^2]$. The resistances to the two adjacent states are determined consecutively, the starting point are the states that are each assigned to a different side of the wall. For each side it is first determined whether the adjacent state is a space, ground or outside temperature, for each case a different transition (air) resistance is summed with the resistance from the *BP_Construction* structure. The resistance and with its related adjacent state are accordingly added to the *m_connections* vector of each *BP_Dep_Class* involved (including itself).

BP_Floor

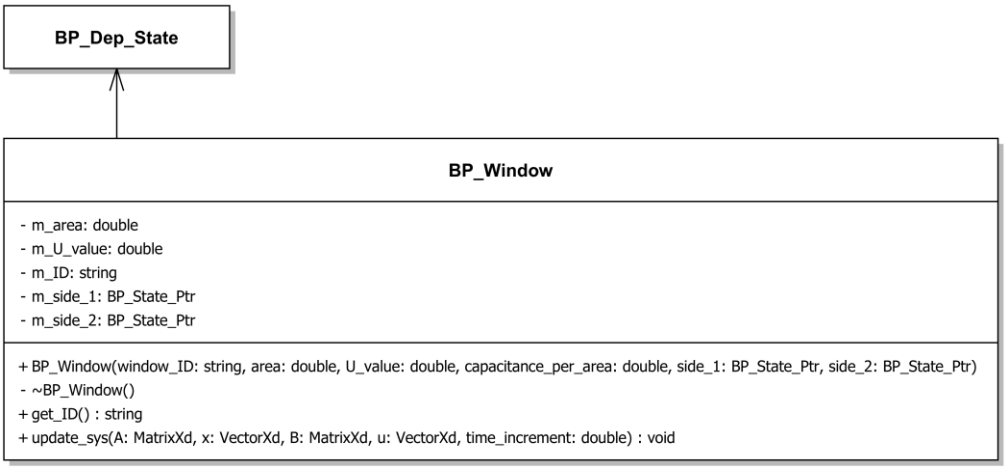
The floor class is identical to the wall class, a distinction has been made as the values of transition resistances are different for horizontal structures. Additionally there may be future implementations in the *update_sys()* function that account for e.g. solar irradiation, which is accounted for differently on walls.



3-14 Class diagram of the *BP_floor* class

BP_Window

This class is inherited from BP_Dep_State, unlike the wall class it is not defined by a construction. Since glass is often not composable, but usually is premanufactured and thus comes with standardised values regarding its thermal properties. These values can be directly stored into the classes member variables.



3-15 Class diagram of the BP_Window class

The member variables of the *BP_Window* class are intended for identification of specific window objects and to determine the values of the base class: *BP_Dep_State*. As such the capacitance is again easily calculated by equation (29). The heat conduction of a sheet of glass is given by a value called the U-value $[W/(K \cdot m^2)]$, the resistance of a sheet of glass towards either side is taken to be half of its total resistance. The resistance is therefore calculated as follows, see equation(30).

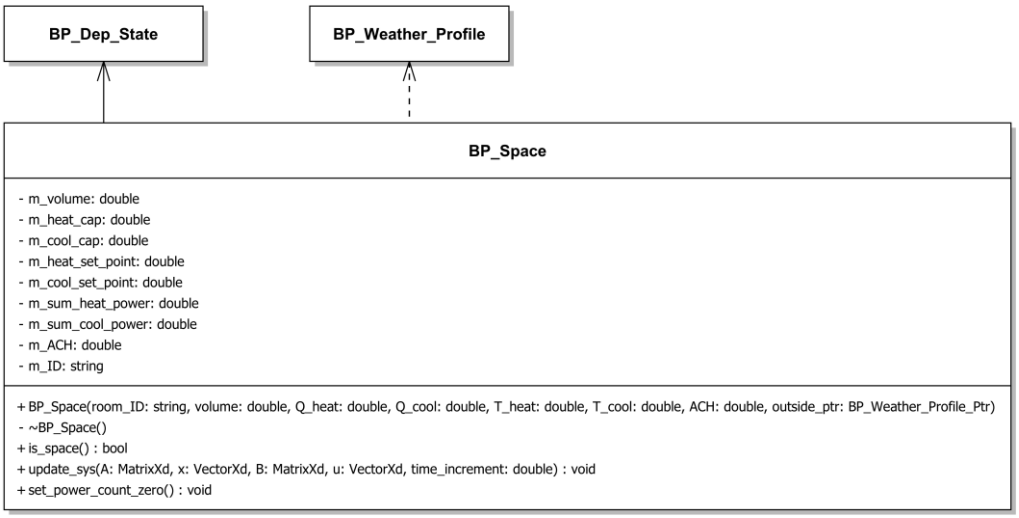
$$R_{side\ i} = \frac{1}{2 \cdot U \cdot A} \tag{30}$$

In the equation $R_{side\ i}$ is the window's heat resistance to side i in $[K/W]$, U stands for the U-value in $[W/(K \cdot m^2)]$, A stands for the window's area in $[m^2]$. All member variables of the base class can be determined using the above, thus the window can be part of the state space system. Windows give a building great sensitivity to solar irradiation, therefore additional member variables like the solar gain factor of a

window will be added when solar irradiation will be supported. The *update_sys()* function, that is currently empty for this class, will then update the state space matrices with the heat flows due to solar irradiation.

BP_Space

The spaces class is inherited from *BP_Dep_State*, it represents a space with all properties required for a thermal simulation. A space is most importantly defined by its volume, its other properties define how and by what mechanisms the space's temperature is influenced.



3-16 Class diagram of the BP_Space class

The member variables of the space class are related to heating and cooling control and ventilation. The capacity and ventilation to the outside temperature depend on the volume of the space. On initialisation of the class, the capacitance in the base class is calculated according equation (31).

$$C_{space} = V \cdot \rho_{air} \cdot C_{air\ specific} \cdot 3.0 \tag{31}$$

In this equation C_{space} is the capacitance of the represented space, V is volume, ρ_{air} is the specific weight of air and $C_{air\ specific}$ is the specific heat of air. The factor 3.0 is a rough estimate intended to take into account the presence of e.g. furniture the space, which would increase the space's capacity.

The resistances between the space and other states is mostly performed during initialisation of walls, floors and windows. This is due to the fact that a space can have multiple connected states, where as e.g. a wall can only have two, i.e. one on each side. However there is initialised one connection to an adjacent state, namely the weather profile. This connection is made to take into account ventilation, which is expressed in a resistance between outside temperature and the spaces temperature in equation (34).

$$Q = \dot{m} \cdot C_{specific} \cdot (T_i - T_e) = \frac{T_i - T_e}{R} \tag{32}$$

$$R = \frac{1}{\dot{m} \cdot C_{specific}} \tag{33}$$

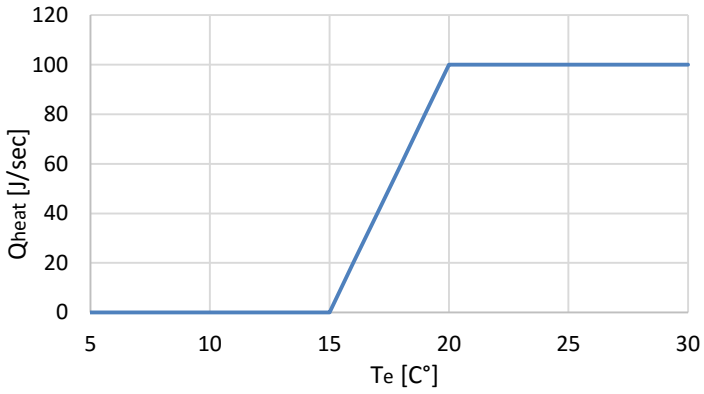
$$R_{outside} = \left(C_{air\ specific} \cdot \left(\rho_{air} \cdot V \cdot \frac{ACH}{3600} \right) \right)^{-1} \tag{34}$$

In the above equations Q stands for heat flux in $[J/sec]$, m for mass flow in $[kg/sec]$, $C_{specific}$ for specific heat in $[J/(K \cdot kg)]$, R for heat resistance in $[K/W]$, ρ for specific weight in $[kg/m^3]$, V for volume in $[m^3]$ and ACH for the number of air changes per hour. Equation (32) equates the heat flux due to a mass flow with the flux due to a resistance, solving the two expressions on the right hand side for R gives the expression in equation (33). Substituting each variable with the space specific properties leads to equation (34).

The only special member function in this class is the *update_sys()* function, which updates the state space system with the heating or cooling power present in the space. This power ($[J/sec]$) is added to the input matrix $\mathbf{B}$ and corresponds to a '1' in the input vector $\mathbf{u}$, which is a dummy state in the vector. This dummy state is necessary because there the relation between the heating or cooling power and the state temperatures is not continuous. This discontinuous relationship is expressed by a so-called P-switch, the expression for heating is given in equation (35).

$$Q_{heat} = \begin{cases} 0 & if \quad T_{set} - T_e \leq 0 \\ \dfrac{T_{set} - T_e}{5.0} \cdot Q_{heat\ max} & if \quad 0 < T_{set} - T_e \leq 5.0 \\ Q_{heat\ max} & if \quad 5.0 < T_{set} - T_e \end{cases} \tag{35}$$

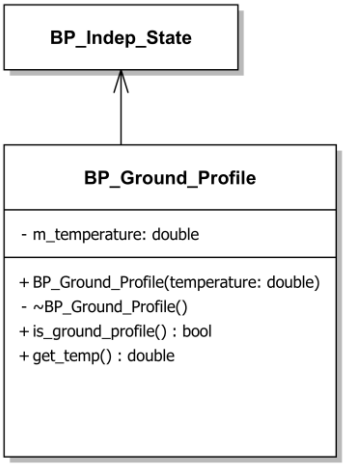
In this equation Q_{heat} is the heating power that will be added to the state space system, $Q_{heat\ max}$ is the maximum heating power available to a space, $T_{set}$is the desired temperature in $[°C]$ and T_e is the outside temperature in $[°C]$. The value 5.0 is the temperature difference in which the heating power ramps up to full power, this value is determined experimentally. The P-switch is intended to keep a space at a constant temperature without it being sensitive for small temperature changes. The experimentally determined value of 5.0 ensures the ramp up is steep enough to be able to sufficiently heat a space but is gradual enough to avoid sensitivities. An example of the P-switch is given in figure 3-17, in which $Q_{heat\ max}$ is set at $100\ kW$ and T_{set} is set at $20\ °C$.



3-17 Graph of the discontinuous function of the P-switch

BP_Ground_Profile

The ground temperature class is inherited from the *BP_Indep_State* class, its only function is to hold the value of a constant temperature value. The ground temperature is constant and does not rely on the season however such a ground temperature profile could be implemented in the future. No particular functions are introduced in this class, the class diagram of the class is depicted in figure 3-18.



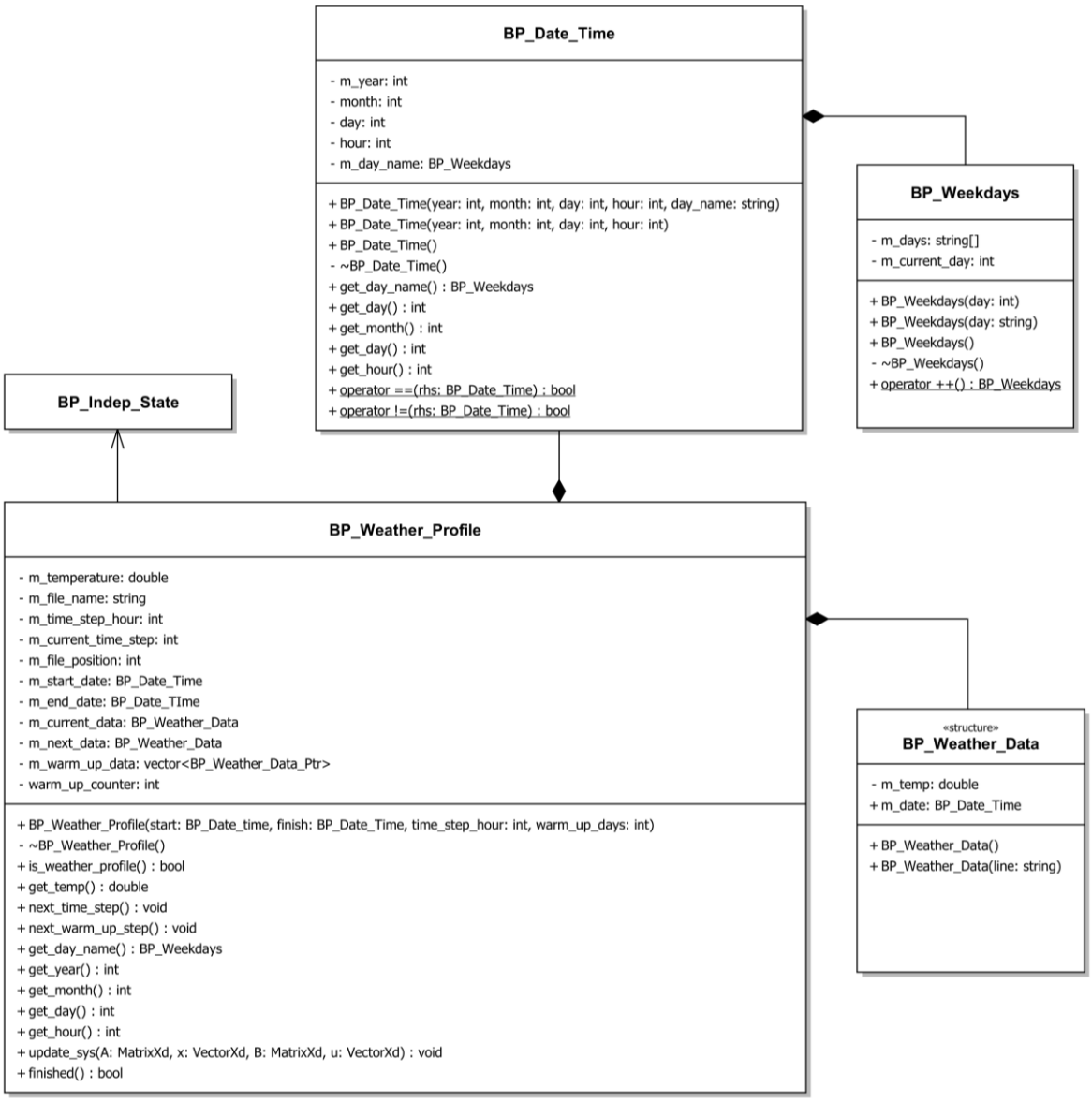
3-18 Class diagram of the BP_Ground_Profile class

BP_Weather_Profile

The outside temperature class is inherited from the *BP_Indep_State* class, its function is to load the correct data from weather files. This data concern date, time and outside temperature that are measured at a location. Currently the data is extracted from files published by the KNMI (royal Dutch meteorology institute), but the *BP_Weather_Profile* class is not dependent on this data source. The class diagram of this class, including classes and structures around it are presented in figure 3-19.

The member variables in the *BP_Weather_profile* class are intended to keep track of the weather file and position in that file, to keep track of time steps, to keep track of the date and time and to store a small amount of weather data for interpolation and a warm up simulation. The *BP_Weather_Data* structure can be initialised with a string from a weather file, i.e. one set of meteorological measurements. The constructor of the structure then extracts the data from that string by means of a tokenizer. Currently temperature, date and time are the only data extracted from the file, however extraction of additional data e.g. solar radiation, is easily implemented. The *BP_Date_Time* class is intended to keep track of the date and the time of the simulation, this class is used to check whether the date and time match the data that is being read from the weather file. Another purpose could be to keep track of what day is simulated, this enables the distinction between e.g. weekdays and weekend to determine occupancy.

Besides overloaded functions from the base class and some getter functions there are functions that control the weather data, i.e. the *next_time_step()* and the *next_warm_up_step()* functions. These stepper functions either interpolate data or they move to new data. Finally, the *update_sys()* function updates the input vector $\underline{u}$ of the state space system with the temperature of the current time step.

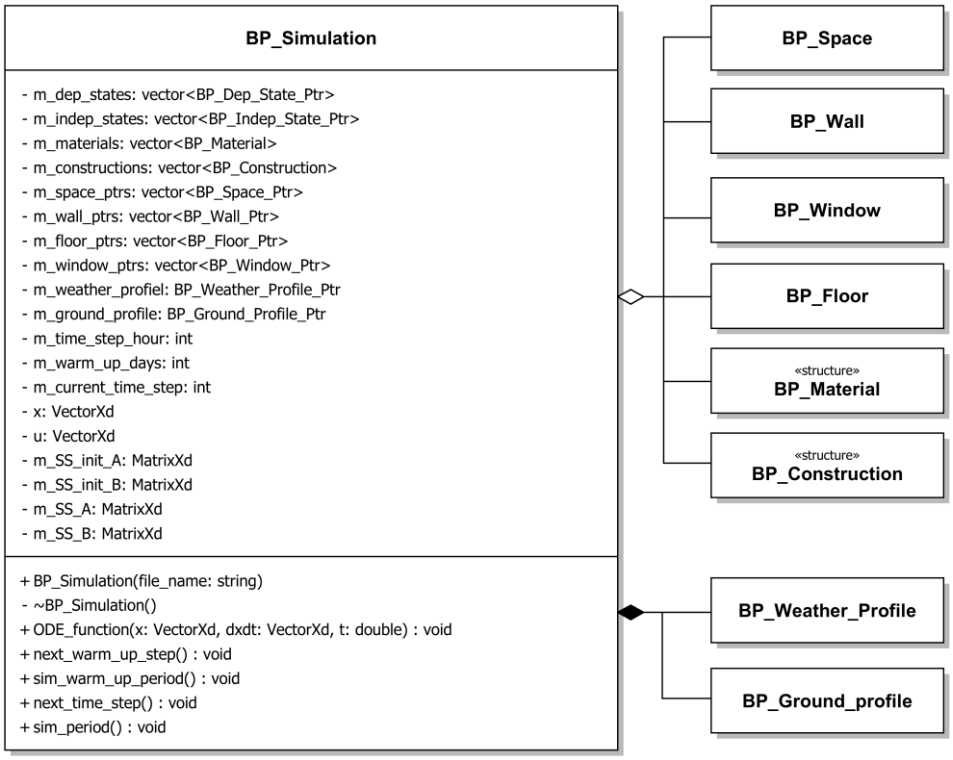


3-19 Class diagram of the BP_Weather_Profile class

BP_Simulation

The *BP_Simulation* class is a composition of multiple *BP_Dep_State* classes and a few BP_*Indep_State* classes. The purpose of the simulation class is to store general building data, simulation data and data concerning the state space system. There are two important functions to the class firstly, the assemblage of the RC-network i.e. the state matrix $\mathbf{A}$, the state vector $\mathbf{x}$, the input matrix $\mathbf{B}$, and the input vector $\mathbf{u}$. Secondly, another function is to step through the simulation time and subsequently to solve the state space system for each time step.

The member variables of this class are related to either the state system or to the simulation. Regarding the state system, this class contains the state matrices and manages all objects of the state classes (held in vectors).



3-20 Class diagram of the BP_Weather_Profile class

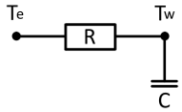
The constructor reads the input file and accordingly initialises states, materials and constructions or it stores some simulation related variables. The matrices and vectors are initialised to their appropriate sizes after the input file has been read.

The member functions of the simulation class control the simulation, for both the warm up period and the simulation period there is a stepper function (*next_time_step()* and *next_warm_up_step()*) and a simulation function (*sim_period()* and *sim_warm_up_period()*). A stepper function steps through one time step, while a simulation function steps through subsequent time steps until the last time step has been reached. The ODE-solver, Odeint, is called in the stepper function by a single line of code. The solver is passed five arguments, the selected solver, the ODE-function (equation (13)), the time increment, a start time and end time. Dummy variables are used for the time related arguments because the system is solved for each time step and no refinement of time steps is required. A refinement of time steps here would mean a division of a time step into multiple time steps, this is undesirable because the state space system cannot be updated during the ODE-solver's operation.

3.4. *Verification of the C++-program*

Case 1, single resistance between two states

The first test case verifies if the implementation of the RC-network into the C++-program is correct. This is done by simulating a first order network, i.e. a state depending only on one other state, see figure 3-12.

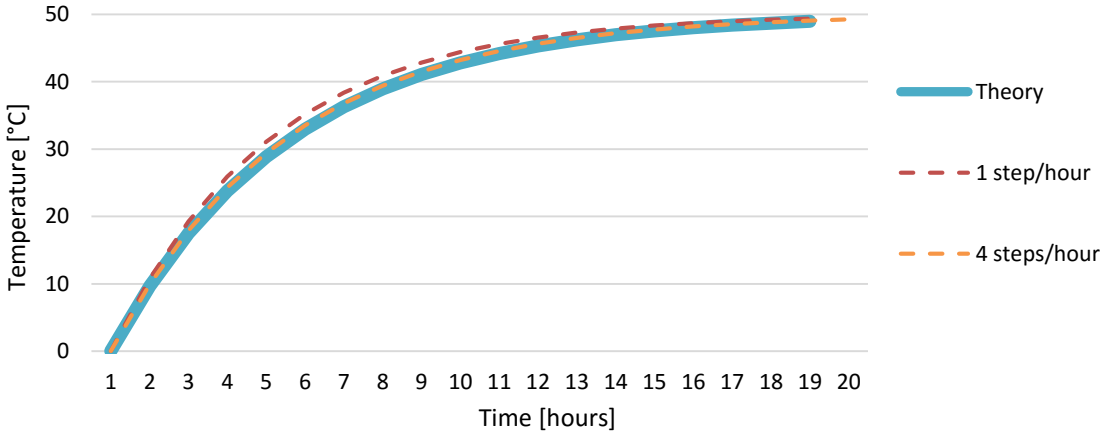


3-21 First order RC-network

The external temperature T_e is in the given example a constant ($50\ °C$), the dependant state temperature T_w starts at $0\ °C$. The resistance R has been set to $4.495 \cdot 10^{-3}\ K/W$ and the capacitance C to $3718 \cdot 10^{3}\ J/K$. The first order network can be verified analytically by the following equation, equation (36).

$$T_w = T_e \cdot \left(1 - e^{-\left(\frac{t}{R \cdot C}\right)}\right) \tag{36}$$

The solver that is used for this specific test case is the Euler method, which has a relatively large error but can with sufficient time steps reach a good approximation. The input file for this problem has been added to Annex 6. The example has been simulated twice, once with one time step per hour and once with four time steps per hour, the results are shown in figure 3-22.

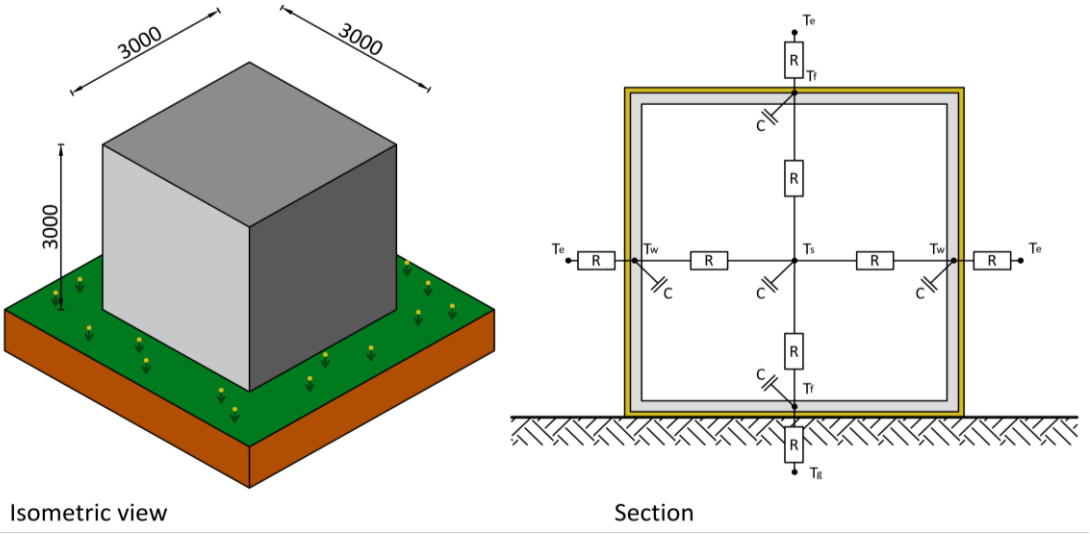


3-22 Simulation results of a first order RC-network solved with the Euler method

The results show that the Euler method can already give a good approximation in a first order network with only one time step an hour. The results show that the program can compose the state space system of a first order RC-network correctly. How the program handles multiple states, heating, ventilation and a varying outside temperature will be tested under the following heading.

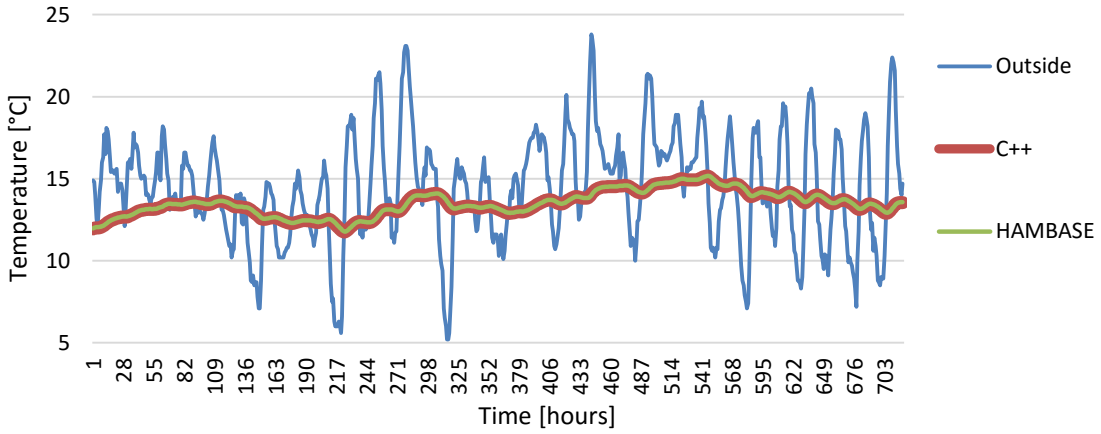
Case 2, concrete box

The second case is a building consisting out of one space of 3 by 3 by 3 metres. The walls and floors of the construction are all made of one construction: $100\ mm$ concrete and $50\ mm$ insulation. The building is simulated with an outside temperature that is measured in September 1985 in De Bilt, Holland. The bottom floor has a constant external temperature of 10 °C, which resembles the ground temperature. The input file for the simulation is given in Annex 7, a visualisation of the building is depicted in figure 3-22.

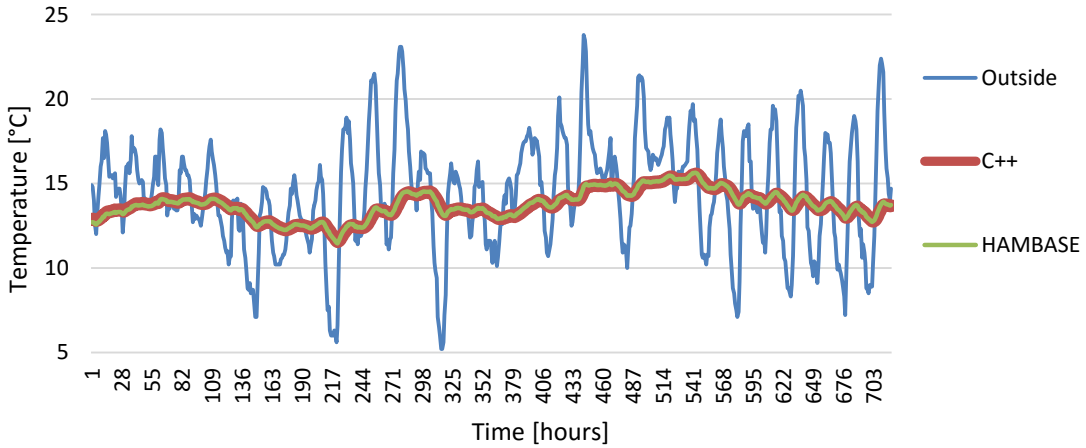


3-23 Visualisation of the concrete box example

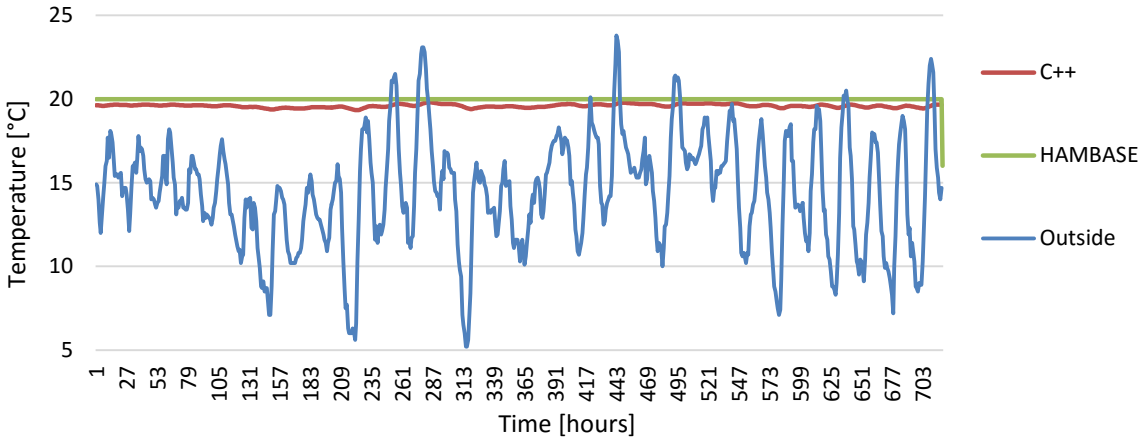
The simulation results cannot be verified by analytical means anymore, therefore the results will be verified by the HAMBASE code in MATLAB. First a high order solver is used, namely the runge_kutta_dopri5 solver. This results in accurate results that can be used to verify the functionality of the program. Four simulations are carried out to test features like outside temperature profile, ventilation and heating in the simulation program, the results are shown in figures 3-24 to 3-29.



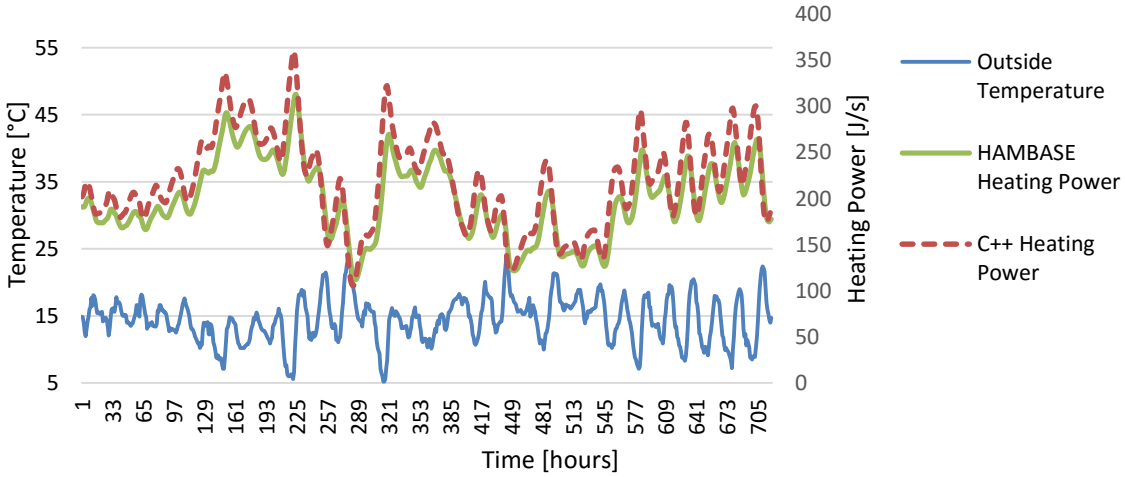
3-24 Space temperature during simulation without ventilation and heating



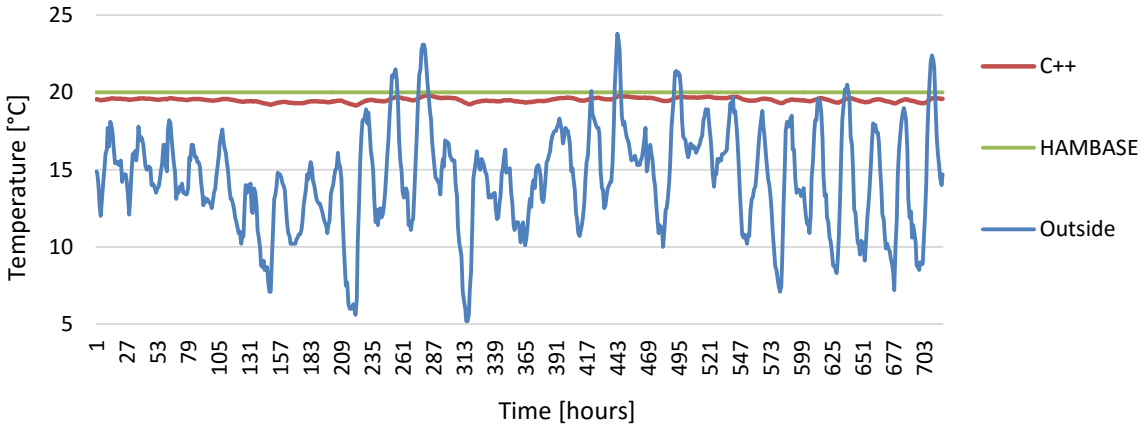
3-25 Space temperature during simulation with ventilation (1 air change per hour) but without heating



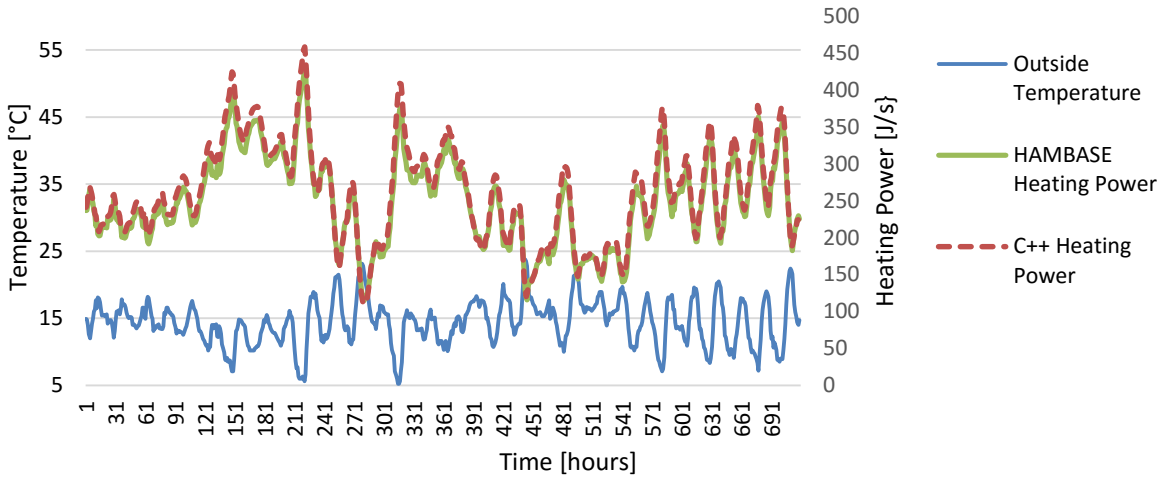
3-26 Space temperature during simulation without ventilation but with heating



3-27 Heating power during simulation without ventilation but with heating ($Q_{max} = 100$ W/m³; $T_{set} = 20$). Cumulative amount of energy: C++: $161\ kWh$; HAMBASE: $144\ kWh$



3-28 Space temperature during simulation with ventilation (1 air change per hour) and heating ($Q_{max} = 100$ W/m³; $T_{set} = 20$)



3-29 Heating power during simulation with ventilation (1 air change per hour) and with heating ($Q_{max} = 100$ W/m³; $T_{set} = 20$). Cumulative amount of energy: C++: $193\ kWh$; HAMBASE: $180\ kWh$

The results of the C++ program show good comparison with the HAMBASE code, the space temperature in both simulations correspond when heating is absent. It is thus concluded that the RC-network has been composed correctly by the C++-program.

When comparing the heating of the space there is a difference to be noted, without ventilation C++ simulates a heating demand that is 11.6% higher than that of HAMBASE, with ventilation this difference is 7.1%. The temperature simulated by the C++-program when the space is heated is however lower than simulated by HAMBASE. The difference in temperature is due to the difference of the used heating switches in both programs, HAMBASE computes an estimate for the heating demand at each time step while the C++-program uses a P-switch. Altogether, the differences in heating demand remain proportionally the same during a simulation. An explanation for the differences has not been found, however the degree of simplicity of the building model and the size of the error do not lead to reasons to question the functionality of the C++-program.

Accordingly, now the functionality of the program has been verified, it is of interest to test the influence of the selected solver. This will be done by carrying out the simulation without heating and ventilation for different solvers and time steps. From the test it has been found that the solvers were unable to compute meaningful results depending on the order of the solver and the number of time steps used for the simulation. The number of time steps required for each solver is presented in Table 3 below.

Table 3 Number of time steps required for each ODE-solver

Solver	Order of the solver	Minimum number of time steps required
Euler	1	6
runge_kutta4	4	5
runge_kutta_dopri5	5	4
runge_kutta_fehlberg78	8	3

Although the results of the example of a concrete box look promising, this is not yet the simulation of a full building. Therefore a selection of a solver cannot be made based on preceding examples, as it should also work for larger buildings. A full building is however not tested in this chapter, but will be in the next chapter. The next chapter presents a method to use a building model in the optimisation toolbox as input for building physics simulations.

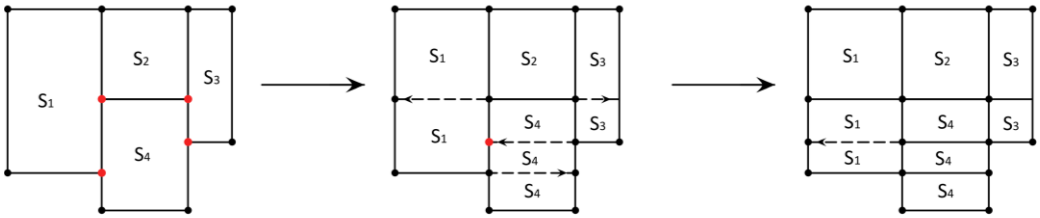
4. Extending the toolbox with BP-analysis

A program that is capable to simulate heat transfer in a building has been developed, yet a means to incorporate the program in the optimisation toolbox is still missing. The program requires an input file in which the different states, e.g. spaces, walls, floors and windows, must be entered manually.

This chapter presents a method that can make the representation of a building conformal, i.e. spaces are divided into smaller cubes so that there are no points intersecting anywhere on or in any cube part of that space. Doing so also splits a space's surface into parts, these parts make sure that a wall is divided at places where multiple adjacent spaces join. A C++ code is presented thereafter, this code can make a building representation conformal and maps relationships neighbouring spaces. Finally an example is given, in which a building in "Movable and Sizable" representation is made conformal by the C++-code and subsequently an input for building physics is generated and simulated.

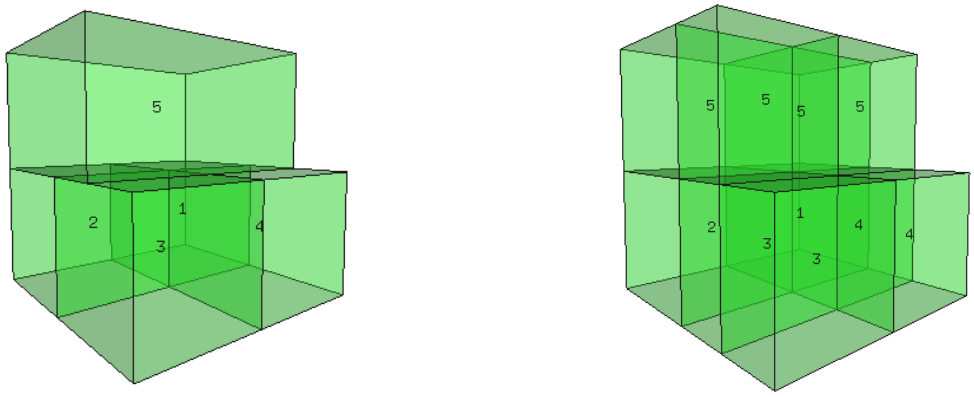
4.1. *Conformal building representation*

A building that consists of multiple spaces is in its basis a concatenation of multiple adjoint spaces. This property of a building consequently forms points and lines at locations where more than two spaces adjoin. Such points and lines are often special lines in computer simulations of buildings like finite element analysis or building physics analysis. They are special because these are also the locations where components in different directions of the simulation model either adjoin or stop.



4-1 Example of four surfaces that are made conformal

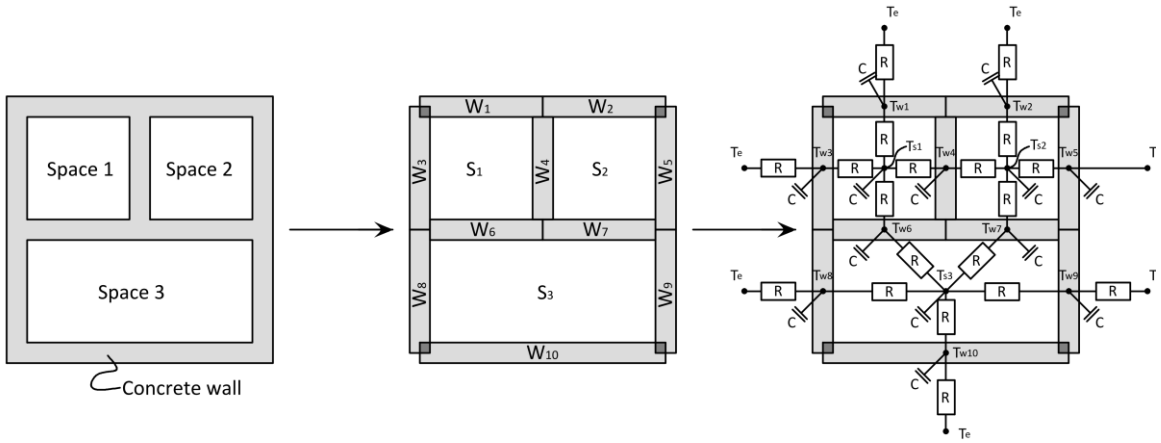
Figure 4-1 shows on the left the floorplan of a building representation in which points exist where more than two spaces adjoin. There are however points in the picture (red) that lie on a continuous wall, division of this wall is necessary so that simulation models can be built properly. A building model in which divisions for such purposes have been applied is here called a conformal building model. The division points are easily identified as they intersect the line of an adjacent surface, by simply checking each point for each line. In the example spaces, rather than walls only, are divided appropriately so that the model is suitable for simulations. An appropriate division in the case of rectangles and cuboids would occur orthogonally, creating new rectangles and cuboids as depicted in figures 4-1 and 4-2. The choice to either divide walls or to divide spaces in a building model is related to the type of simulation model it is intended for.



4-2 Example of a building design (left) and its conformal representation (right)

When making a building conformal, each divisions may create new points of intersection (figure 4-1 middle). This requires a subsequent division that accounts for the newly created intersection, creating a conformal building is thus recursive. This recursive problem may be solved by iteration, i.e. checking each point in the representation after divisions have been performed until no more intersections exist. Another approach would be to directly check a newly created point and accordingly dividing if necessary, the latter itself is recursive. Solving the problem recursively is likely to be the most efficient approach to tackle the problem, however recursion is a notorious cause of problems in programming and is often hard to visualise for a programmer. Section 4.2 will discuss how recursion is used to in the C++ code that has been developed to make building models conformal.

The need for a conformal building representation arises from the fact that elements like walls and floors in a building physics model stop at a location where more than two spaces adjoin. This is due to the fact that a wall acts as a link between the temperature states of the adjoining spaces, this has been visualised in figure 4-3.



4-3 Visualisation of translation from building design to BP-model to RC-network

A simple building design is given in the above picture, although the wall is all one material this is not the case when considering a building physics model. The building physics model requires information about the spaces at either side of a wall to be able to build an RC-network. The monolithic concrete wall can be split up into part that can be handled by a building physics simulation program.

Making a building representation conformal does however not add the information about which spaces are on either side. This can be done by checking the location of a wall with the location of all spaces in the model when this information is required. Another approach is to create an integral system of spaces, surfaces and lines of the model, in which each line is coupled to the spaces it is in and vice versa each space to the lines it consists of. The latter has been implemented in the C++ code and will be discussed in more detail in section 4.2.

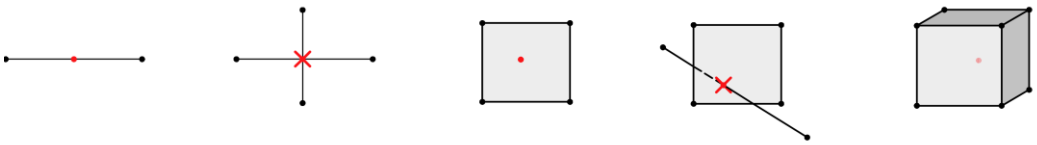
Methodology

Making a building model conformal requires first of all checks that are used to the identify points that should divide a part in the building model. In case of an orthogonal building model there are four types of parts that can be used in a building model a vertex (0D), a line (1D), a rectangle (2D) and a cuboid (3D). Each higher dimension is a composition of multiple lower order parts, thus a line can be related to a cuboid but a cuboid must consist of multiple lines, see Table 4.

Table 4 Relation between different parts in an orthogonal building model

Part	Number of parts needed			
	Vertex	Line	Rectangle	Cuboid
Vertex	1	-	-	-
Line	2	1	-	-
Rectangle	4	4	1	-
Cuboid	8	12	6	1

Each of these parts should be checked for conformity, because of the relations in the above table these checks can be performed by checking all lines, rectangles and cuboids for intersecting vertexes. The relations between parts also allow to check for overlaps in which case a line intersects either another line or a rectangle. The cases that then need to be checked are illustrated in figure 4-4.

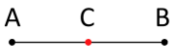


Case 1 Case 2 Case 3 Case 4 Case 5

4-4 Base cases that need to be checked when making a space conformal

The above cases can be checked by means of vector calculus, vectors are computed by means of the vertices of which a part consists. A general check for each case has been defined by using simple operations like point products, cross products and magnitudes of vectors. A point product between two vectors: $\mathbf{v_1} \cdot \mathbf{v_2}$ computes a scalar and can be used to calculate the angle between two vectors, however when the scalar is zero then the two vectors orthogonal. A cross product between two vectors: $\mathbf{v_1} \times \mathbf{v_2}$ computes a vector that is normal to both vectors, however when cross product results in a zero vector then the two vectors are parallel. In an orthogonal building model this is all that is required to check the cases, as follows.

Case 1 – Point on a line



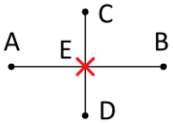
4-5 Point on a line

Three vectors are computed, $\mathbf{v_1}$ from *AB*, $\mathbf{v_2}$ from *AC* and $\mathbf{v_3}$ from *BC*, see figure 4-5. Two checks determine if the point lies on the line, if equation (37) holds then the two vectors are collinear and the point either lies on line *AB* or in its extension.

$$\mathbf{v_1} \times \mathbf{v_2} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{37}$$

Accordingly vectors $\boldsymbol{v_2}$ and $\boldsymbol{v_3}$ are in x-, y- and z direction checked whether they have the same direction, i.e. if the signs of the two vectors in any direction are the same then the point does not lie on the line. If this check remains false for each direction then the point lies on line *AB*.

Case 2 – Intersection between two lines



4-6 Intersection point of two lines

Three vectors are computed, $\mathbf{v_1}$ from *AB*, $\mathbf{v_2}$ from *CD* and $\mathbf{v_3}$ from *AC*, see figure 4-6. Three checks determine whether vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ intersect. First, if equation (37) holds for both $\mathbf{v}_1 \times \mathbf{v_2}$ and $\mathbf{v}_1 \times \mathbf{v_3}$ then it is proven that the two vectors are collinear, and if an intersection then exist this will be found by case 1. Second, if equation (37) holds for $\mathbf{v}_2 \times \mathbf{v_3}$ then it is proven that the two vectors are parallel and these exists no point of intersection. Third, if equation (38) holds then it is proven that the two vectors are skew and thus not coplanar, there then does not exist a point of intersection

$$\mathbf{v_3} \cdot (\mathbf{v_1} \times \mathbf{v_2}) \neq 0 \tag{38}$$

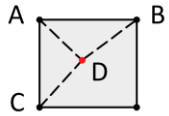
If all of the preceding checks remain false, then there exists a point of intersection. This point is found by the following equation, equation(39).

$$E = A + s \cdot \mathbf{v_1} \tag{39}$$

In which s is a scalar calculated by equation (40). The scalar s is also used to determine whether point *E* lies on line *AB* by checking if it is smaller than 1.

$$s = \frac{\|\mathbf{v_2} \times \mathbf{v_3}\|}{\|\mathbf{v_1} \times \mathbf{v_2}\|} \tag{40}$$

Case 3 – Point on a rectangle



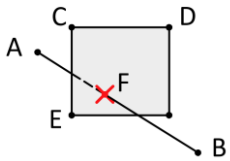
4-7 Point on a rectangle

Five vectors are computed, $\mathbf{v_1}$ from *AB*, $\mathbf{v_2}$ from *AC*, $\mathbf{v_3}$ from *AD*, $\mathbf{v_4}$ from *BD* and $\mathbf{v_5}$ from *CD*, see figure 4-6, where *C, D* and *E* are three arbitrary vertices of the rectangle. Two checks are performed, firstly if equation (41) holds true then the point does not lie on the plane of the rectangle.

$$\mathbf{v_3} \cdot (\mathbf{v_1} \times \mathbf{v_2}) \neq 0 \tag{41}$$

If above equation holds false, then it should be determined whether the point lies on the rectangle. This can be determined by checking vectors $\mathbf{v_3}$, $\mathbf{v_4}$ and $\mathbf{v_5}$ in x-, y- and z direction if they have the same direction, i.e. if the signs of the two vectors in any direction are the same for all vectors then the point does not lie on the rectangle. If this check remains false for each direction then the point does lie on the rectangle.

Case 4 – Intersection of a line on a rectangle



4-8 Intersection point of a line on a rectangle

Four vectors are computed, $\mathbf{v_1}$ from *AB*, $\mathbf{v_2}$ from *AC*, $\mathbf{v_3}$ from *CD*, and $\mathbf{v_4}$ from *CE*, see figure 4-8. One check is performed to determine if the line intersects the plane of the rectangle, if equation (42) holds true then the line is either parallel to the plane or coplanar. When the line is coplanar and intersecting the rectangle then this will be found by either case 3 or case 2.

$$\mathbf{v_1} \cdot (\mathbf{v_3} \times \mathbf{v_4}) = 0 \tag{42}$$

If the above equation remains false, then there exists a point on the plane of the rectangle where the vector $\mathbf{v_1}$ intersects. This point is computed by equation (43)

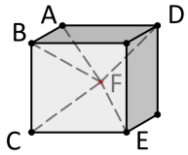
$$F = A + s \cdot \mathbf{v_1} \tag{43}$$

In which s is a scalar calculated by equation (44). The scalar s is also used to determine whether point *F* lies on line *AB* by checking if it is smaller than 1.

$$s = \frac{\|\mathbf{v_2} \times \mathbf{v_3}\|}{\|\mathbf{v_1} \times \mathbf{v_2}\|} \tag{44}$$

If the line intersects the rectangle's plane at point *F*, then it should still be checked if the point lies on the rectangle, this check can be performed as discussed with case 3.

Case 5 – Point inside a cuboid



4-9 Point inside a cuboid

This case is checked by computing five vectors, $\mathbf{v_1}$ from *AF*, $\mathbf{v_2}$ from *BF*, $\mathbf{v_3}$ from *CF*, $\mathbf{v_4}$ from *DF* and $\mathbf{v_5}$ from *EF*, see figure 4-9, where *A* until *E* are five arbitrary vertices of the cuboid. This case can be determined by checking if vectors $\mathbf{v_1}$, $\mathbf{v_2}$ $\mathbf{v_3}$, $\mathbf{v_4}$ and $\mathbf{v_5}$ have the same direction in either x-, y- or z direction, i.e. if the signs of the two vectors in any direction are the same for all vectors then the point does not lie on the rectangle. If this check remains false for each direction then the point does lie on the rectangle.

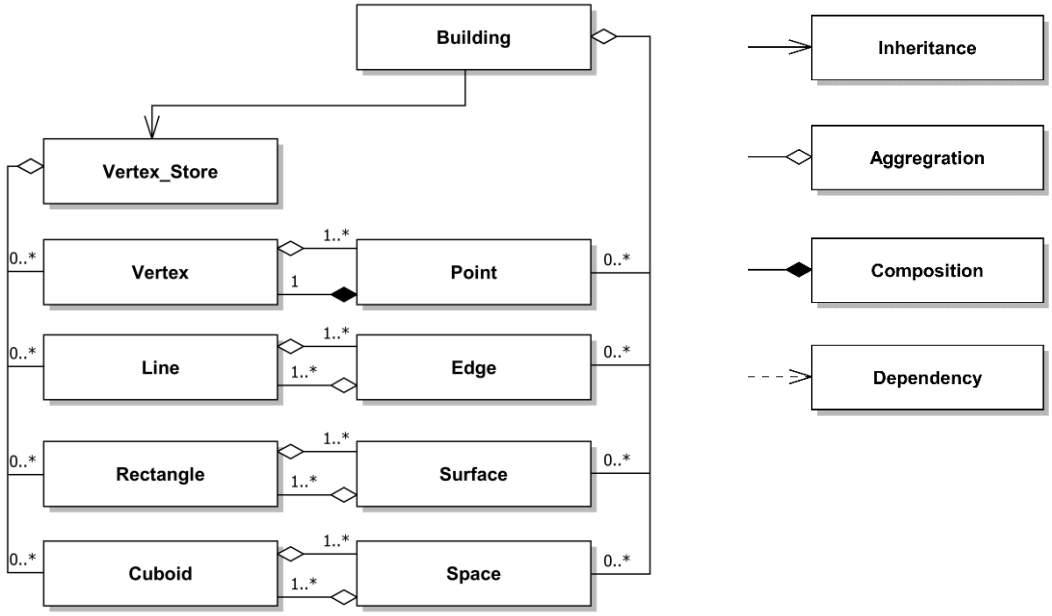
An orthogonal building model can be made conformal by identifying the above cases in the model and dividing either walls or spaces. The next section presents a C++ code in which a class structure is presented that holds data on the building side e.g. spaces, surfaces and edges and data on the conformal side e.g. vertices, lines, rectangles and cuboids. The two data sides remain separate from each other but they share information between them, a rectangle can this way be related to the spaces on both its sides. This makes the C++ code suitable to translate a building model into a building physics model since relations between conformal model parts (walls/rectangles) and building parts (spaces) need to be merged in the building physics model.

4.2. *Conformation of a building model in C++*

This section clarifies the C++ code that has been written to transform a building model into a conformal building model. This is done by translating a building model into geometric entities like cuboids, rectangles, lines and vertices. These geometric entities are split while keeping their relation with the original building model, e.g. a space can be split into multiple cuboids therefore the space consists out of multiple cuboids and each of the cuboids is related to the space. The code has been divided into two sides, a building side where data related to the building are handled and a conformal side where data regarding the geometric entities are handled.

Main structure of the code

The conformation program has been written in C++ using an object oriented approach, the class diagram of the main structure of the code is given in figure 4-10.The diagram shows on the left side classes related to the conformity and on the right side classes related to the building. The *Building* class is the main interface of the code, in which an input is read and stored and functions to make a building conformal are implemented. The *Building* class also holds the functions related to the generation of simulation output.

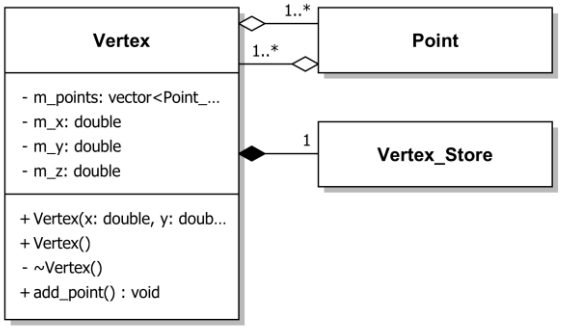


4-10 Class diagram, showing the main structure of the C++ code

The diagram above shows general relations between classes, more specific relations are not shown, for example the relations listed in Table 4 are not shown in the picture. The depicted classes and the more specific relations to other classes will be discussed in more detail in the remainder of this section. The C++ code has been added to the annexes in the back of this thesis, Annex 8.

Vertex

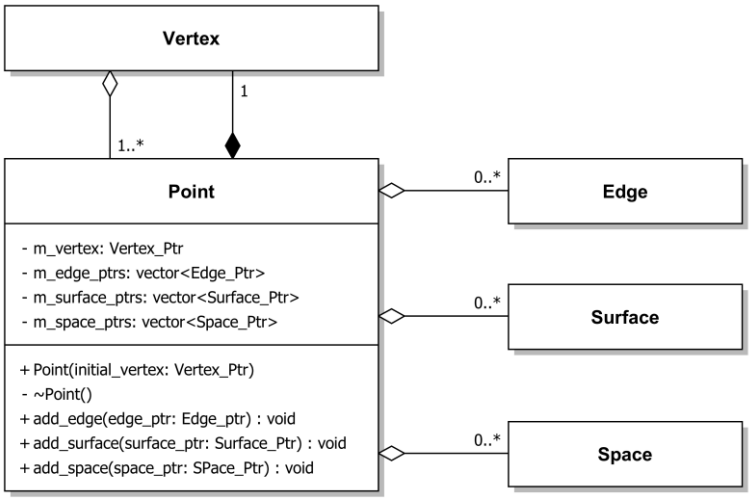
The *Vertex* class will represent all vertices in the conformal building model. The class is in its basis a set of x- y and z- coordinates, but also holds a relationship with the building side: the *Point* class. The class diagram of the *Vertex* class is given in figure 4-11



4-11 Class diagram of the *Vertex* class

Point

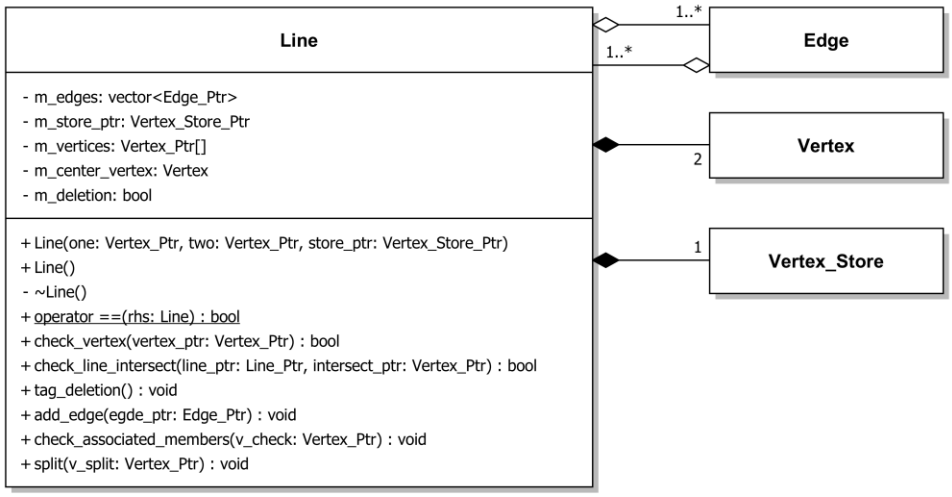
The *Point* class will represent all points in the non-conformal building model. It is composed out of one instance of the *Vertex* class and is related to the either the space, surfaces or edges it is part of. The member functions are intended to initialise such relationships to the class. Figure 4-12 shows the class diagram of the *Point* class



4-12 Class diagram of the *Point* class

Line

The *Line* class represents all lines in the conformal building model, it is defined by two vertices, figure 4-13. An instance of this class accounts for relationships with the building side by storing pointers instances of the *Edge* class. A pointer to the *Vertex_Store* class has also been added, this enables a line to split into new lines and subsequently add them to the vertex store. The splitting of a line makes the original line redundant, therefore a tag for deletion has been added to the *Line* class. This tag is necessary because there may still be instances of other classes that point towards the instance of the class that is to be deleted.

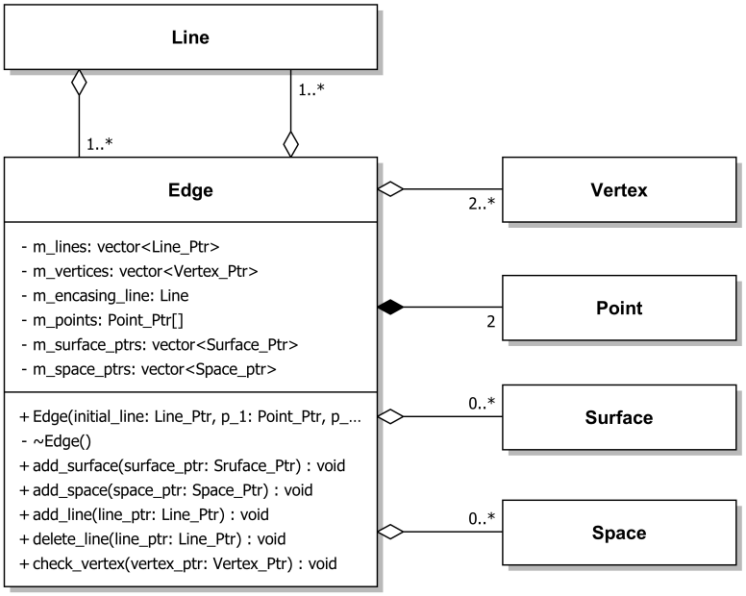


4-13 Class diagram of the *Line* class

The first member functions of the *Line* class check if an instance affect other lines or vertices. The overloaded == operator is used in the *Vertex_Store* to make sure no duplicate lines are added to the store. The two check functions check for case 1 and case two as described in section 4.1. The *split()* function splits a line at the given vertex, accordingly after splitting there are two instances of the *Line* class added to the Vertex_Store. When a line has been split, then also the *check_associated_members()* function is called, this function will check the vertex that split the line with all the building member associated with the line.

Edge

The *Edge* class represents all edges in the non-conformal building model, it is in defined by one or more instances of the *Line* class, see figure 4-14. An instance of this class is composed of two points and its relations to the surfaces and spaces of which it is a part are aggregations. All unique vertices of the lines in *m_lines* are also stored, to quickly asses if a vertex is already part of an instance of the *Edge* class.

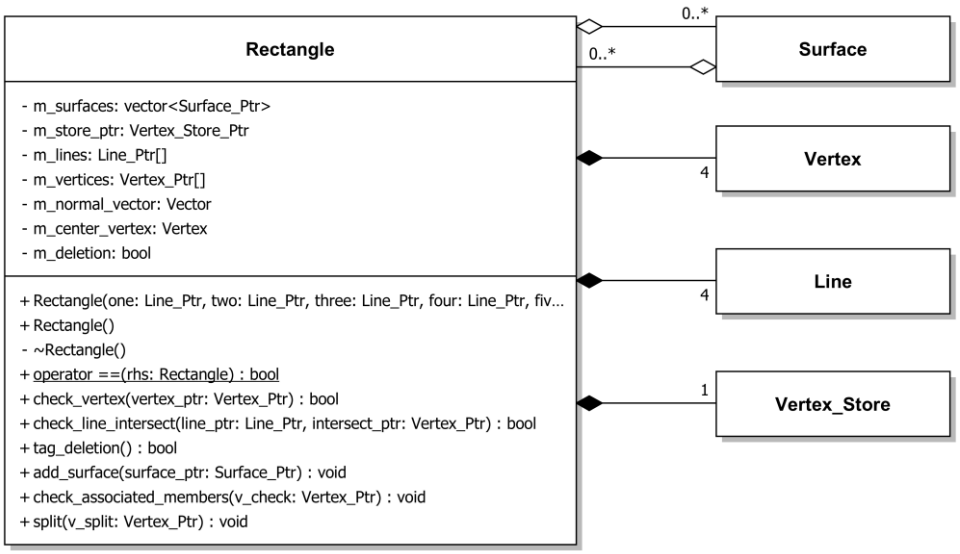


4-14 Class diagram of the *Edge* class

The three add functions are used to keep the relations of an instance with instances of other classes up to date. The delete function is used after a *Line* instance has been split, a line is split when the *check_vertex()* function determines that a *Vertex* instance lies on the edge but has not been added to the *m_vertices* vector yet.

Rectangle

The *Rectangle* class represents all rectangles in the conformal building model, it is composed out of 4 vertices and 4 lines, see figure 4-15. Relationships with the building side are stored in a vector with pointers to *Surface* instances and similar to the *Line* class a deletion tag and a pointer to the *Vertex_Store* instance in which it is stored are here also added for splitting purposes.



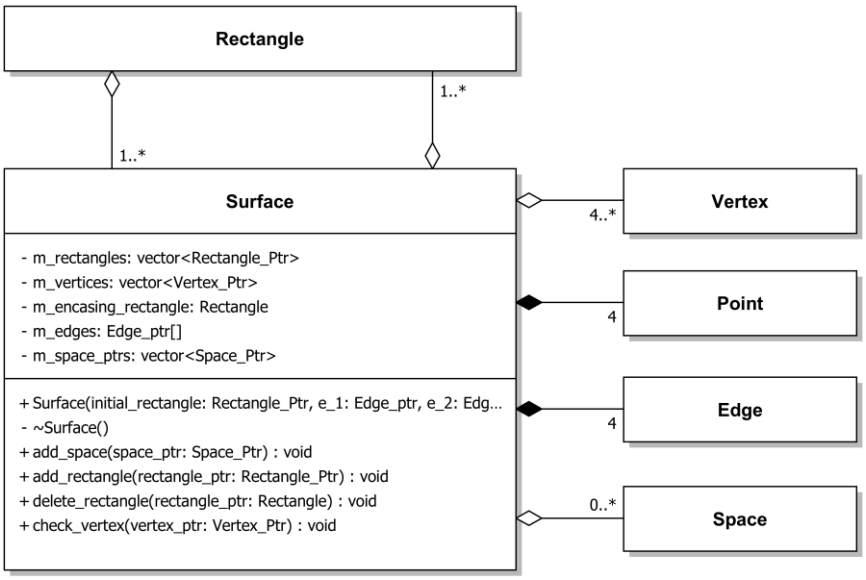
4-15 Class diagram of the *Rectangle* class

The overloaded *==* operator can compare two instances of the *Rectangle* class and determine whether they are coincident. This function is used in the *Vertex_Store* class to prevent duplicates to be added to an instance of the store class. The two check function check case 3 and case 4 as discussed in section 4.1. The *split()* function splits a rectangle at the given vertex, accordingly after splitting there are multiple new instances of the *Rectangle*, *Line* and *Vertex* classes added to the Vertex_Store. When a rectangle has been split, then also the *check_associated_members()* function is called, this function will check the vertex that split the rectangle and the new vertices with all the members (spaces, surfaces and edges) associated with the rectangle.

Surface

The *Surface* class represents all surfaces in the non-conformal building model, it is defined by one or more instances of the *Rectangle* class, see figure 4-16. An instance of this class is composed of four points and four edges, its relations to the spaces of which it is a part are aggregations. All unique vertices of the rectangles in m_*rectangles* are stored to be able to quickly asses if a vertex is already part of an instance of the Surface class.

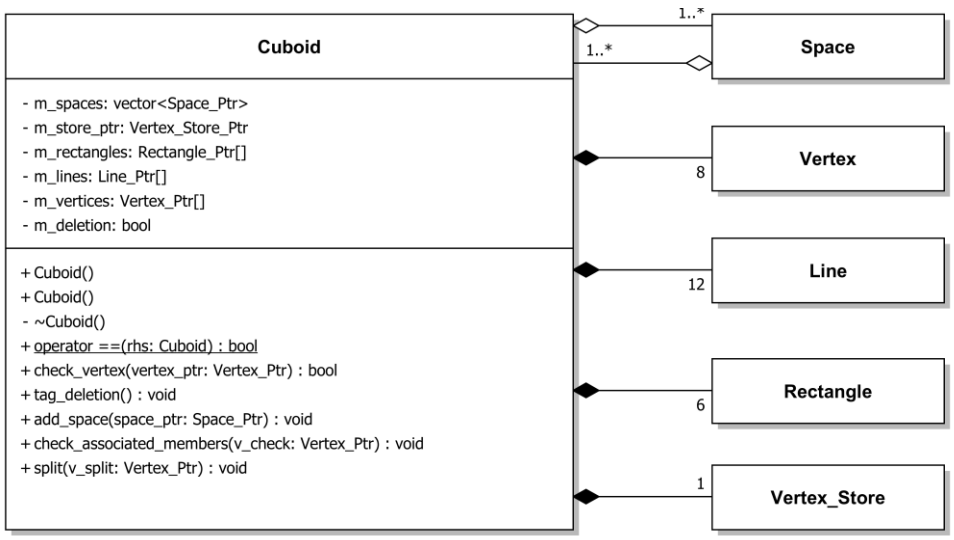
The two add functions are used to keep the relations of a *Surface* instance with instances of other classes up to date. The delete function is used after a *Rectangle* instance has been split, a rectangle is split when the *check_vertex()* function determines that a *Vertex* instance lies on the surface but has not been added to the *m_vertices* vector yet.



4-16 Class diagram of the *Surface* class

Cuboid

The *Cuboid* class represents all cuboids in the conformal building model, it is composed out of six rectangles, twelve lines and eight vertices, see figure 4-174-15. Relationships with the building side are stored in a vector with pointers to *Space* instances and similar to the *Line* and *Rectangle* classes a deletion tag and a pointer to the *Vertex_Store* instance in which it is stored are here also added for splitting purposes.

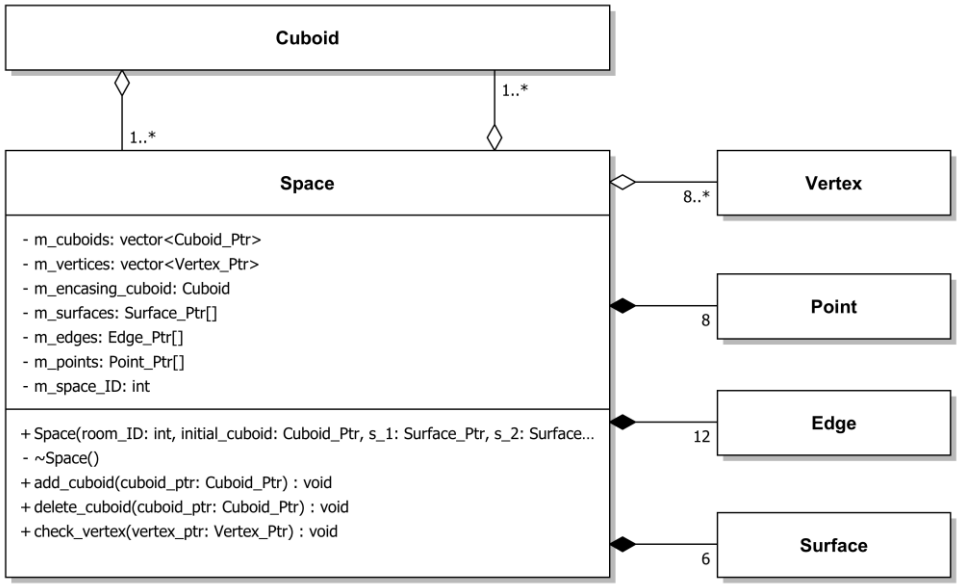


4-17 Class diagram of the *Cuboid* class

The overloaded == operator can compare two instances of the *Cuboid* class and determine whether they are coincident. This function is used in the *Vertex_Store* class to prevent duplicates to be added to an instance of the store class. The check function checks case 5 and discussed in section 4.1. The *split()* function splits a cuboid at the given vertex, accordingly after splitting there are multiple new instances of the *Cuboid*, *Rectangle*, *Line* and *Vertex* classes added to the Vertex_Store. When a cuboid has been split, then also the *check_associated_members()* function is called, this function will check the vertex that split the cuboid and the new vertices with all the members (spaces, surfaces and edges) associated with the split cuboid.

Space

The *Space* class represents all spaces in the non-conformal building model, it is defined by one or more instances of the *Cuboid* class, see figure4-18. An instance of this class is composed of eight points, twelve edges and six surfaces. All unique vertices of the cuboids in m_*cuboids* are stored to be able to quickly asses if a vertex is already part of an instance of the Space class.

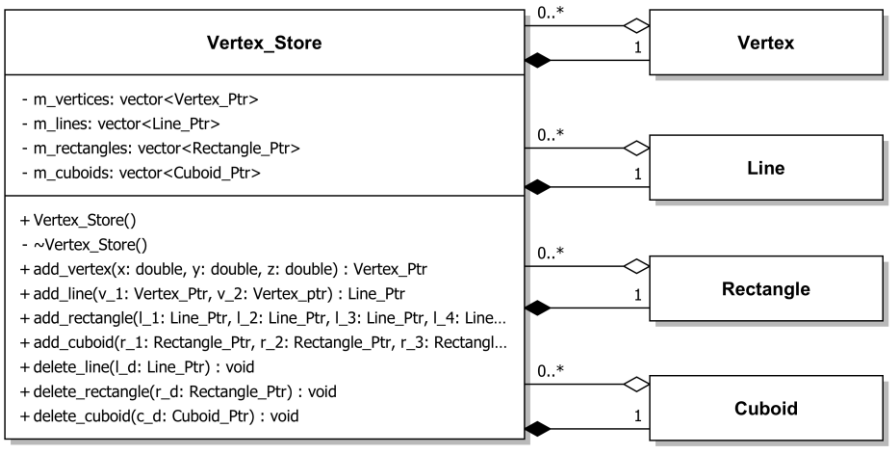


4-18 Class diagram of the *Space* class

The *add_cuboid()* function is used to keep the relations of a *Space* instance with instances of other classes up to date. The delete function is used after a *Cuboid* instance has been split, a cuboid is split when the *check_vertex()* function determines that a *Vertex* instance lies in the space but has not been added to the *m_vertices* vector yet.

Vertex_Store

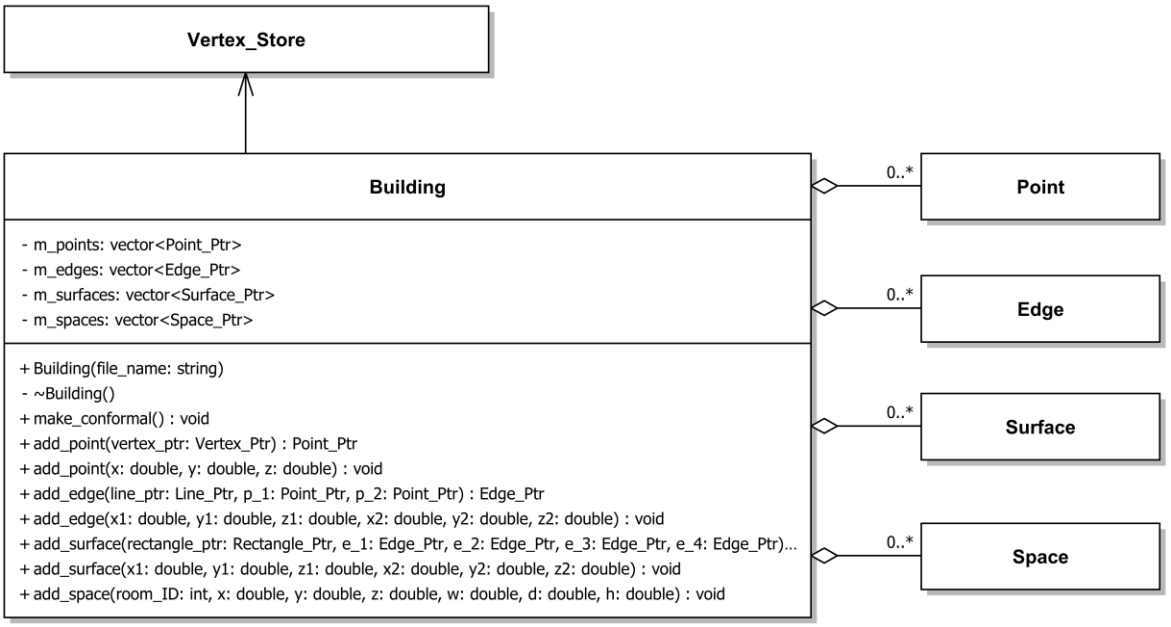
The *Vertex_Store* is purely a store for all vertices, lines, rectangles and cuboids in the conformal building model. It is therefore an aggregation of the *Vertex*, *Line*, *Rectangle* and *Cuboid* classes, see figure 4-19. The purpose of the class is to store instances of the mentioned classes, to prevent duplicates to be added to the store and to remove redundant instances. This purpose is fulfilled by the add and delete functions, where an add function first checks if a class instances is not already in the store before adding it to the store.



4-19 Class diagram of the *Vertex_Store* class

Building class

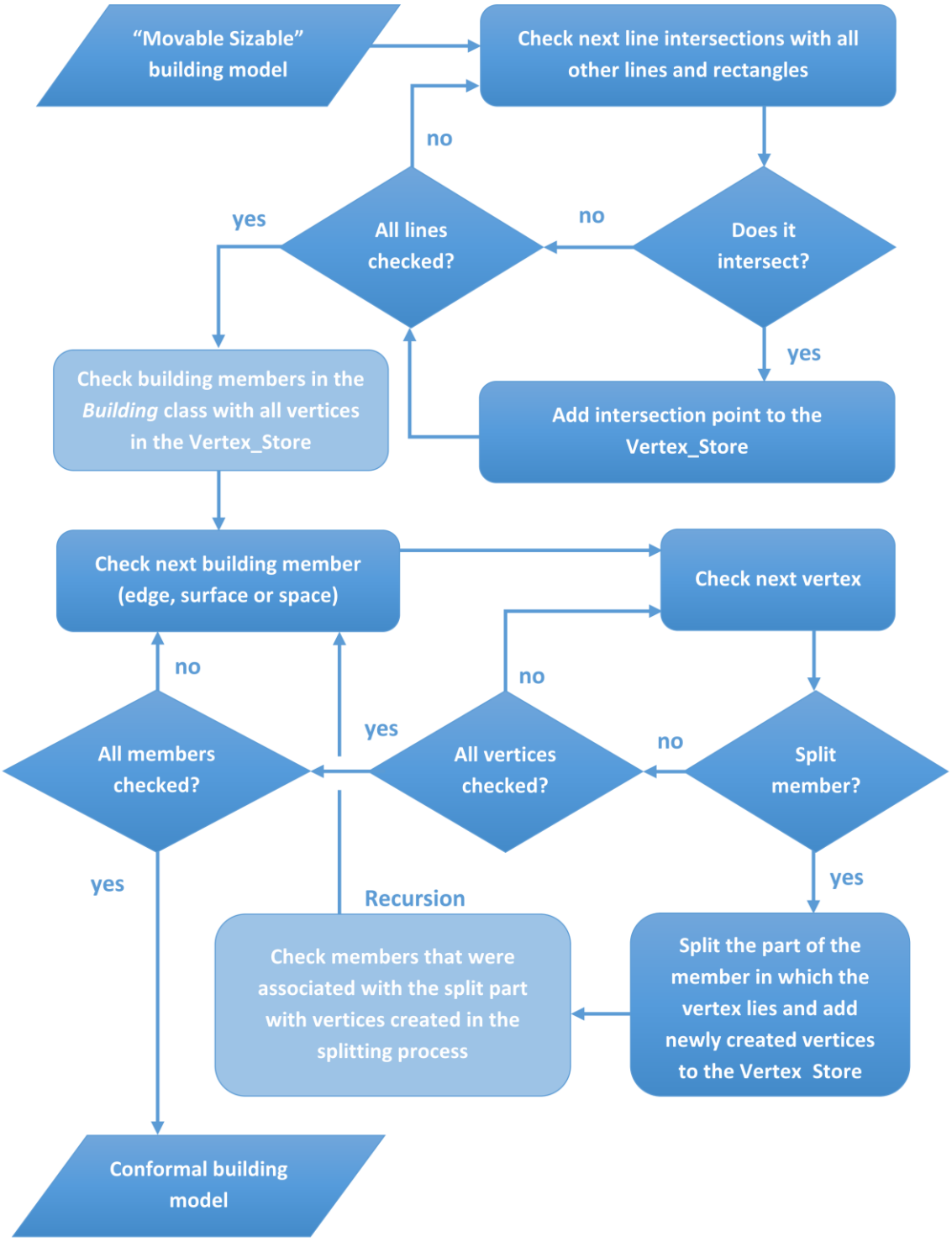
The *Building* class represents the non-conformal building model, i.e. spaces, surfaces of spaces, edges of spaces have not been divided. The class is an aggregation of *Point*, *Edge*, *Surface* and *Space* classes, see figure 4-20. The class inherited from the *Vertex_Store* class, and as such the *Building* class also stores the conformal building model. The class's main purpose is to map relations between the conformal and non-conformal building model, i.e. a space consists of multiple cuboids and a cuboid is part of a space.



4-20 Class diagram of the *Building* class

The add functions in the *Building* class are used in the initialisation of an instance of the class. Initialisation is done by an input file that uses the "Movable Sizable" representation for the building model. For every space in the model there are eight *Point*, twelve *Edge*, six *Surface* and one *Space* instances initialised. Accordingly, for each of the initialised classes, either a *Vertex*, *Line*, *Rectangle* or *Cuboid* is added to the base class, which is the *Vertex_Store* class. The add functions also make sure that the appropriate information about relations between the conformal side and building side are added to each added class.

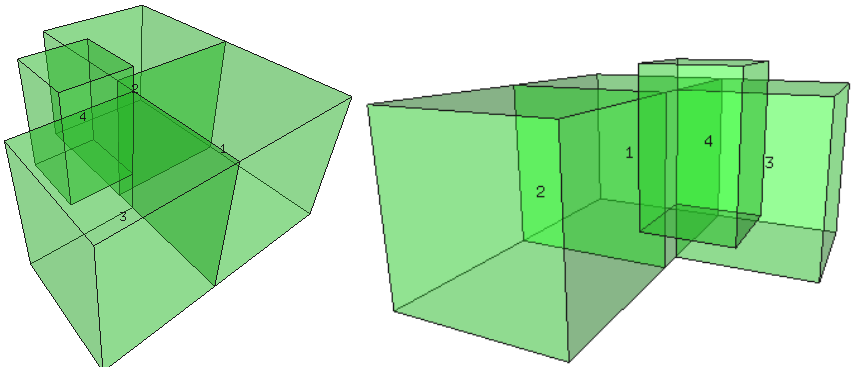
Finally, the *make_conformal()* function will divide every space, surface and edge into multiple cuboids, rectangles and lines. The process that this function follows is depicted in the flow chart in figure 4-21, the function first finds all intersection points of lines with other lines and rectangles and subsequently adds these points as vertices to the instance of the *Vertex_Store* class. All building members, i.e. edges, surfaces and spaces are subsequently checked with all vertices in the vertex store. During the checking of vertices it may occur that either a line, rectangle or cuboid is split. A splitting invokes a new iteration that checks all building members (edges, surfaces and spaces) that were associated with the split part, this is recursion.



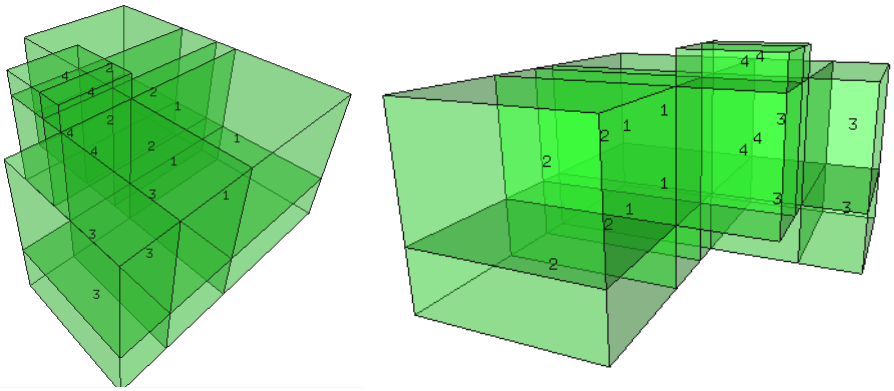
4-21 Flow chart of the conformation process

Verification

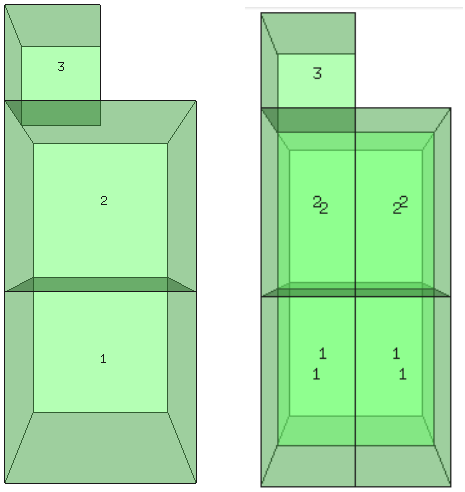
A small test cases was made to verify if the conformation code works correctly. The input and output of the test case are visualised in figure 4-22 and 4-23 respectively. Another case to test if the program can correctly handle recursive properties of conformation is presented in figure 4-24. From the presented test cases it is concluded that the conformation code works correctly.



4-22 Input model of the test case



4-23 Output of the test case, a conformal building model



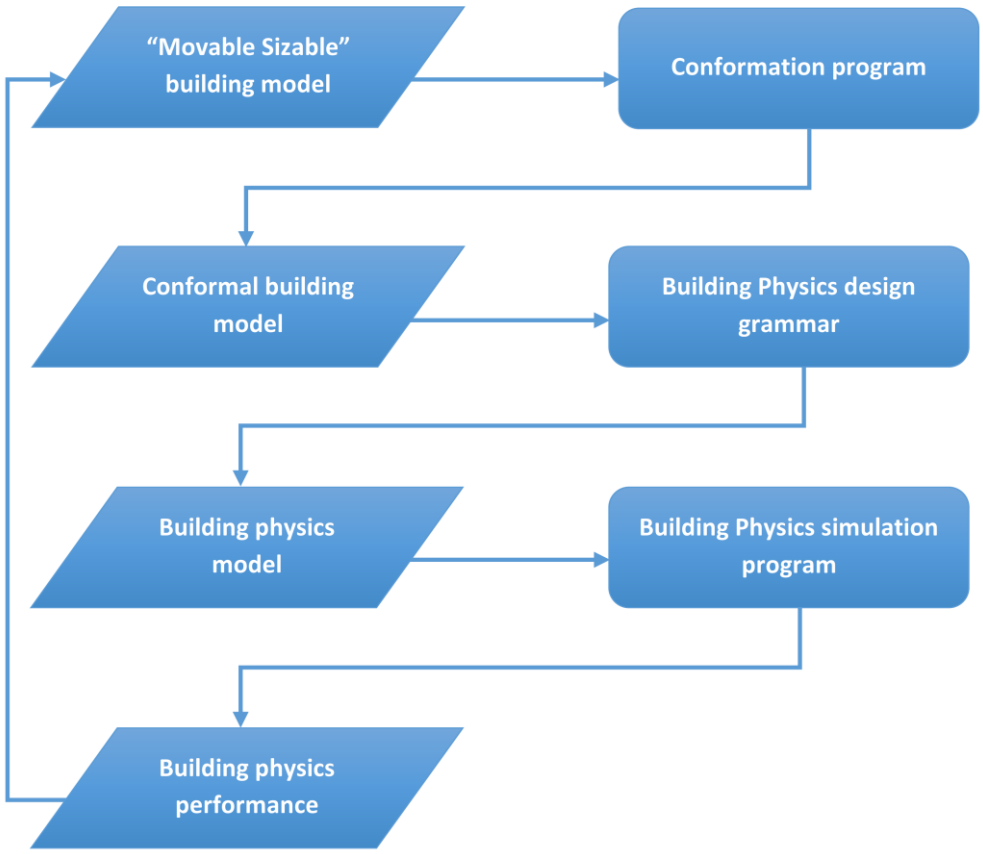
4-24 Input (left) and output (right)

Building physics input

This section has described how the C++ code (Annex 8) creates a conformal building model. However it has not yet been clarified how the conformal building is used to generate input for building physics analysis. This is part of the C++ code, but it will be discussed in the next section.

4.3. *Automated building physics analysis of building models*

The ultimate goal of the presented building physics program and conformation program is to enable automated building physics simulation and analysis. So far only separate steps in the automated process have been illustrated. This section will illustrate the complete process from a “Movable Sizable” representation to a building physics performance. An overview in the form of a flow chart is depicted in figure 4-25, the flowchart illustrates how automated building physics analysis will be performed in the toolbox



4-25 Flow chart of the automated building physics analysis

The flowchart shows how building physics performances of a “Movable Sizable” building model can be determined by running model conformation, a design grammar and a building physics simulation. Accordingly these performances may be used to modify the building model and iterating the process as such in order to increase the total building performance. The conformation and building physics simulation steps have been discussed in this thesis, however the building physics design grammar is yet to be discussed.

Building physics design grammar

The *Building* class as discussed in section 4.2 has an additional function regarding the building physics model. Namely a *make_BP_input()* function, see Annex 8, which translates instances of the *Space* class and members of the *Rectangle* class into building physics input. Spaces are easily added to the building physics model, since only their ID and volume are extracted from the building model. Adding information of rectangles to the building physics model is more complicated. Since it should first be determined whether a rectangle could represent either a wall, or floor, subsequently it should be determined which spaces are adjacent to the wall. Determining if a rectangle is a wall or floor can be done by checking if the normal of the rectangle is vertical, if it is then the rectangle represents a floor. Checking the spaces to which a rectangle is associated allows to determine whether the rectangle should be added to the building physics model, namely if there are no associated spaces then it should not become a wall or floor. If there is one space associated with a rectangle then the rectangle is an exterior wall or floor and has a space on one side and a weather

profile on the other side (walls and floors with $z \neq 0$) or a ground profile on the other side (floors with $z = 0$).

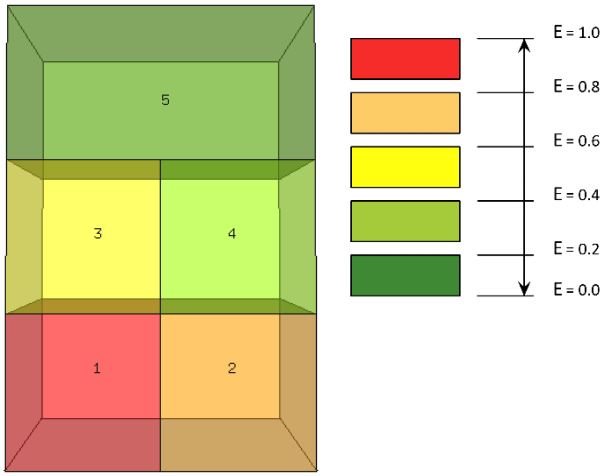
The above only determines the geometric information of a building physics model, properties like material, construction types, heating and ventilation are still missing from the model. The missing information is added by design rules, which together form a design grammar. A design grammar defines the properties of spaces, walls and floors based on design rules, which can be simple like: All rooms have a heating capacity of $100\ W$. Grammars can however be much more complicated to create more sophisticated or more realistic building physics models. A building physics design grammar for the toolbox will not be determined in this thesis, however for demonstration purposes there will be defined some rudimentary design grammars.

The remainder of this section will present several examples in which the automated design process is performed. Starting with some rudimentary examples that have intuitive solutions and progressively moving on to more complex building designs.

Performances in the examples will be expressed in the required amounts of energy per volume to cool and heat a space. These performances will be visualised by means of a colour mapping for spaces, see figure 4-26. The visualisation maps each space with a colour depending on their performance (equation (46)) relative to the maximum ($\overline{Q_{max}}$ $[J/m^3]$) and the minimum ($\overline{Q_{min}}$ $[J/m^3]$) energy performance per space volume (equation (45)) throughout the building.

$$\bar{Q}_i = \frac{Q_{heat,i} + Q_{cool,i}}{V_i} \qquad [J/m^3] \tag{45}$$

$$E_i = \frac{\bar{Q}_i - \overline{Q_{min}}}{\overline{Q_{max}} - \overline{Q_{min}}} \qquad [-] \tag{46}$$

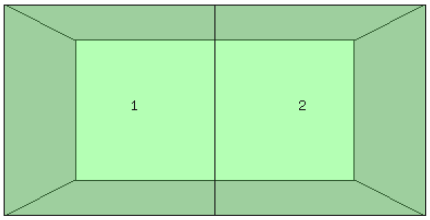


4-26 Visualisation of building physics performance of spaces. Red (1) requires the most energy, green(5) the least, intermediate performances are visualised by orange(2), yellow(3) and lime (4).

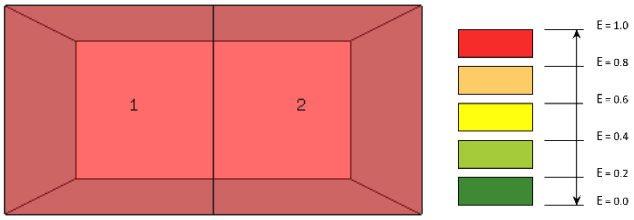
The examples have all been run with the same building physics settings and two different design grammars have been defined and used. The simulation period is the complete year of 1988 and the ground temperature is set to $10\ °C$. The simulations started with six warm up days and took four time steps an hour, the runge_kutta_dopri5 solver has proven to be sufficient for the simulations. Two constructions are defined a wall and a window, the window has a U-value of $1.2\ W/(m^2 \cdot K)$ and a capacitance of $30000\ J/(m^2 \cdot K)$. The wall construction consists of 50 mm of insulation () and 100 mm of concrete. The constructions are assigned by two grammars, grammar 1 only assigns the wall construction and grammar 2 sequentially assigns a wall construction to two successive exterior vertical rectangles and then a window construction to the next exterior vertical rectangle. Grammar two thus assigns a window to a third of all the exterior rectangles in the conformal building model.

Example 1 – Two spaces

The first example is a building that consists of two spaces that are joined together on the ground floor, see figure 4-27. Grammar 1 is used, the example is symmetric and should thus generate symmetric results. Figure 4-28 shows the result of the analysis, it proves to be symmetric thus both spaces perform equally bad.



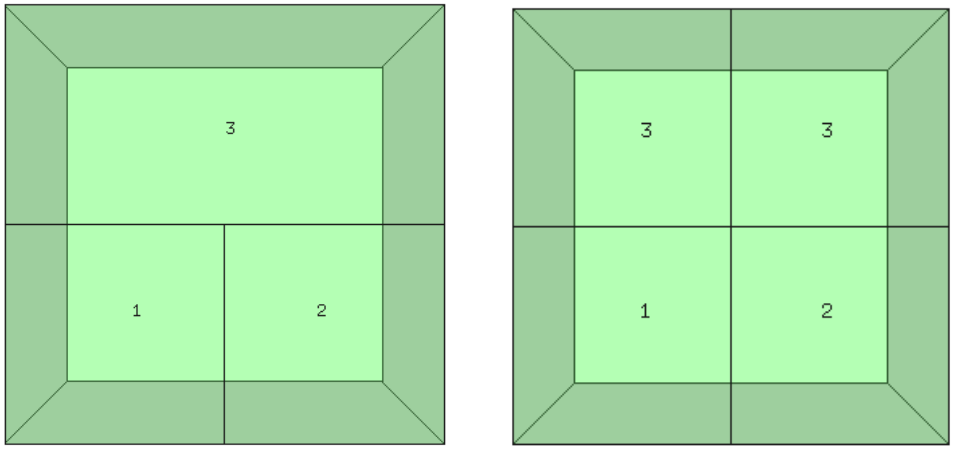
4-27 Visualisation of the building of example 1



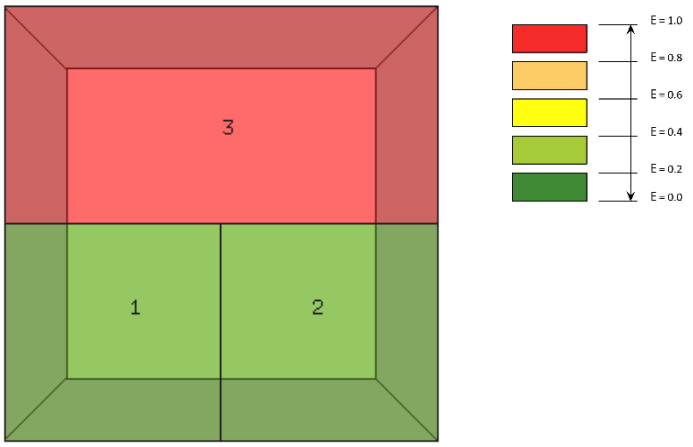
4-28 Visualisation of the space performances in the building of example 1

Example 2 – three spaces

A space is now added atop of the building of example 1, see figure 4-29. Grammar 1 is still used and thus the result is still expected to be symmetric about the separation wall between space one and two. Figure 4-30 shows the analysis result, it proves to be symmetric spaces one and two perform equally. The space on top is the worst performing, which can be explained by the fact that it is exposed to the exterior with a rather large surface area in comparison with the two bottom space.



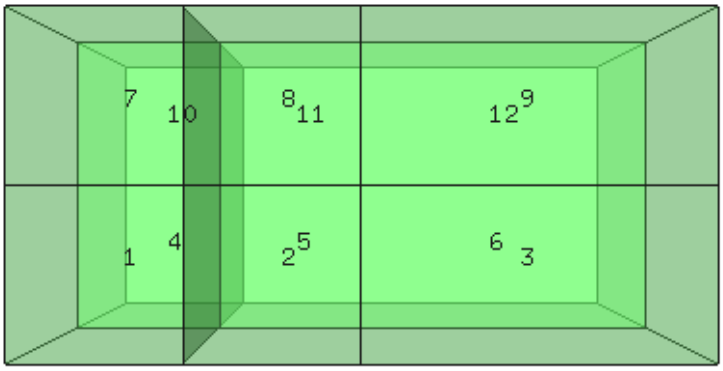
4-29 Visualisation of the building of example 2, together with its conformal model



4-30 Visualisation of the space performances in the building of example 2

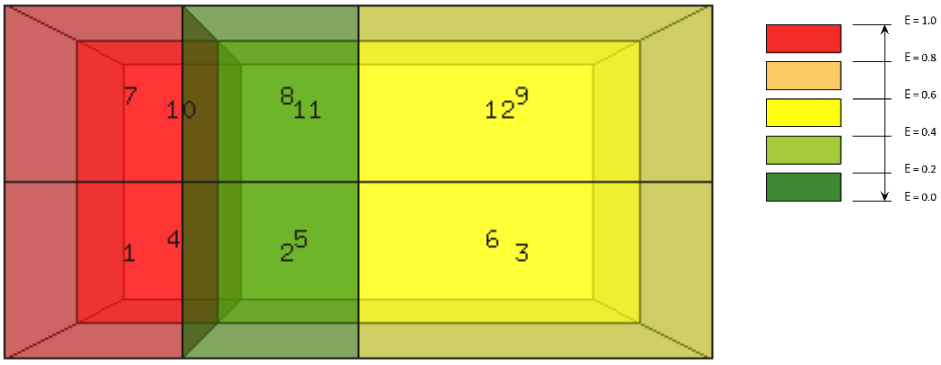
Example 3 – twelve spaces

A cuboid building of twelve spaces, some of which larger than the other is analysed in this example, see figure 4-31. The simulation has been performed twice, for the first simulation grammar 1 is used to generate the building physics model and for the second grammar 2.



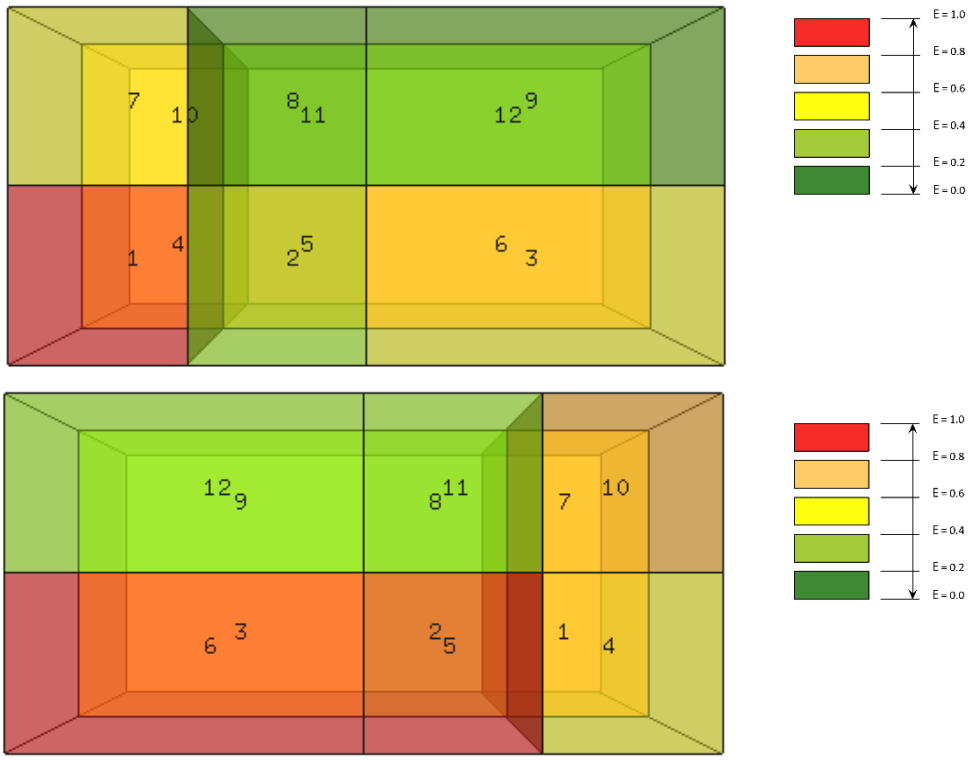
4-31 Visualisation of the building of example 3

Figure 4-32 depicts the analysis result, it shows that the small spaces on one head end perform the worst and the larger spaces perform intermediate. This is surprising at first, since the volume of the smaller spaces is relatively large in proportion with its exposed surface compared to the larger spaces. However when considering the whole of spaces, 1, 4, 7 and 10 and the whole of space 3, 6, 9 and 12 then these relations turn around, which explains the result.



4-32 Visualisation of the space performances in the building of example 3 with grammar 1

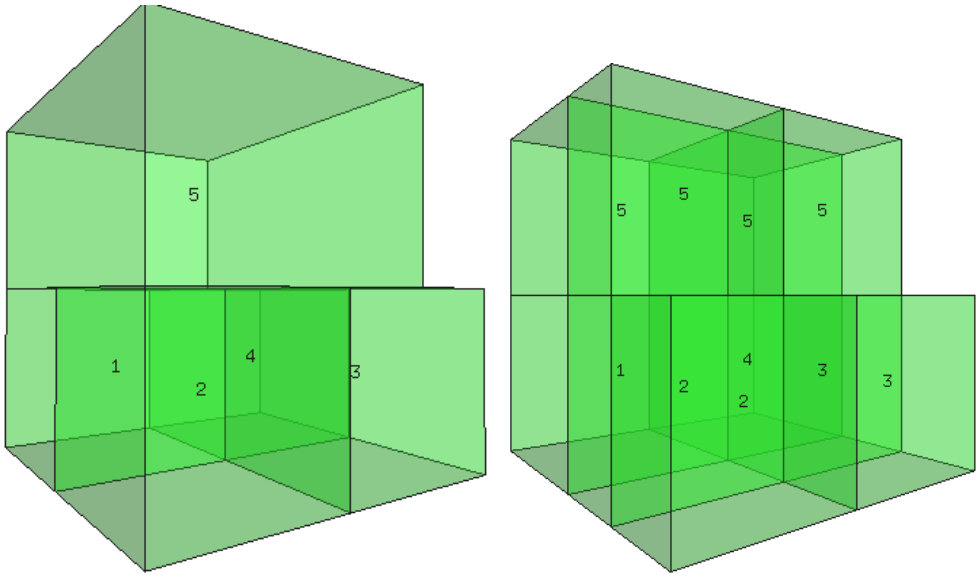
Figure 4-33 depicts the result for the design by grammar 2, it shows more differentiation in the performances throughout the building. The result is however rather meaningless because the relationship with the presence of windows is not given. However the difference in results do illustrate the influence that design grammars can have in the automated design of buildings.



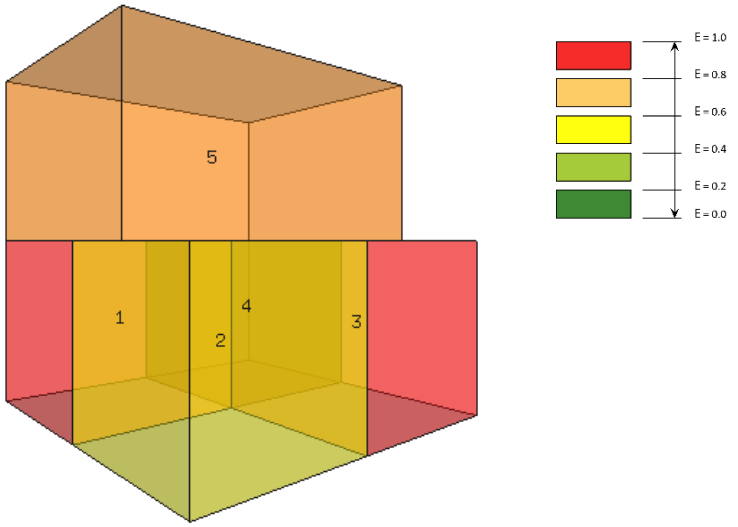
4-33 Visualisation of the space performances in the building of example 3 with grammar 2

Example 4 – Arbitrary building design

For this example an arbitrary building design was designed, see figure 4-34. The building physics model has been generated by grammar 2. The result of the analysis is shown in figure 4-35, again the results are not meaningful with regard to the building spatial design. However this example is a strong show case of what the developed tool is capable of.



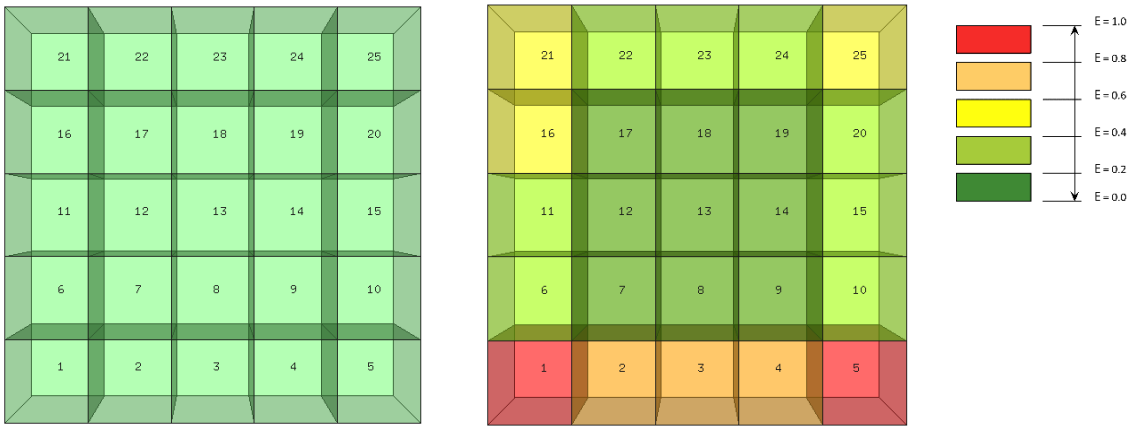
4-34 Visualisation of the building of example 4



4-35 Visualisation of the space performances in the building of example 4

Example 5 – Apartment building

An apartment building has been defined for the final example, see figure 4-36. The building physics model is generated by grammar 2. The results are have also been depicted in figure 4-36 and show that spaces in the middle of the building outperform the spaces on the sides. Additionally it can be concluded that spaces at the corners generally perform the worst and spaces on the ground floor generally perform worse. The design grammar did have a large influence on the differentiation in the results, there is only one space which jumps out performance wise.



4-36 Visualisation of the building of example 5 (left), and the results of the analysis (right)

5. Conclusions and recommendations

5.1. *Summary*

This thesis presents two methods and three tools that have been developed for use in multi-disciplinary building optimisation. The first method aims to increase the variation in building designs in an optimisation task by considering a super structure free and a super structured design space at the same time during one optimisation process. Two different building representations were defined, each suitable for one of the two considered design spaces. A tool has been developed to convert each building representation into the other and vice versa, this tool is envisioned to enable both design space types to be used during one optimisation task.

The second method aims to automatically asses building designs on their performance with respect to building physics, i.e. required energy for climate control in a building. This automated process uses a simulation program to determine performances of spaces in the building model. A simulation tool was developed to simulate the heat balance of a building and the heating and cooling of spaces. The simulated energy required for heating and cooling of spaces is used to assess the performance of spaces with the respect to the building. A 3D-conformation tool that is computationally fast was developed to make the representation of a building design compatible with the building physics simulation tool.

5.2. *Conclusions*

Two building representations have been defined, a “Movable and Sizable” representation that is suitable for super structure free design spaces and a “Super Cube” representation that is suitable for super structured design spaces. Algorithms have been developed to transform each representation into one and other, and these algorithms have been verified for successful operation for overlaps in spaces, non-connected spaces, truncation errors, alterations in space identification, and fragmented spaces.

From a research on building physics simulation programs it was concluded that the development of a simulation tool would be the best approach to analyse the defined building representations. Accordingly a tool using a thermal resistor-capacitor network model was developed, this tool computes a state space system that is solved for each time step in the simulation.

A tool to conform a building model to be compatible to the building physics too has been developed. This conformation tool has been used to run the automated building physics analysis on several test cases.

5.3. *Recommendations*

The presented methods and tools for multi-disciplinary building optimisation have been tested and verified individually. However their application to optimisation itself has not yet been tested, the next stage for multi-disciplinary building optimisation is therefore to incorporate the methods and tools in the optimisation toolbox. The work presented in this thesis is in continuation of an existing toolbox for spatial building design optimisation for optimal structural design performance and thus the tools will be implemented in this toolbox.

Topics for the toolbox and multi-disciplinary optimisation that still need research are as follows. Grammars for both building physics and structural design need to be defined in order to generate realistic building designs. Modification rules for spatial modification should be researched, current modification cannot evolve building designs into a building with more floors.

Optimisation using the super structured and super structure free design space approaches should be combined as proposed in this thesis. Case studies should be performed using the super-structure free approach, the super structured approach and the combined approach to research multi-disciplinary optimal spatial building design. Accordingly the results of the case studies will be used to find design relations in the form of design rules. Applications for the found design rules should be developed for early building spatial design.

6. References

Attia, S., Beltrán, L., Herde, A. De, & Hensen, J. (n.d.). " Architect Friendly ": a Comparison of Ten Different Building Performance Simulation Tools. In *Proceedings of Building Simulation* (pp. 204–211). Glasgow, Scotland.

Baldock, R., & Shea, K. (2006). Structural Topology Optimization of Braced Steel Frameworks Using Genetic Programming. In *Intelligent Computing in Engineering and Architecture, 13th {EG}-{ICE} Workshop* (pp. 54–61). http://doi.org/doi:10.1007/11888598

Bandaru, S., & Deb, K. (2015). Temporal Innovization: Evolution of Design Principles Using Multi-objective Optimization. In *8th International Conference of Evolutionary Multi-Criterion Optimization* (pp. 79–93). http://doi.org/10.1007/978-3-319-15934-8

Blom, K. Van Der, Boonstra, S., Hofmeyer, H., & Emmerich, M. T. M. (n.d.). A Super-Structure Based Approach for Building Spatial Designs. In *submitted for EC-COMAS 2016* (p. -). Crete, Greece.

Caldas, L. (2008). Generation of energy-efficient architecture solutions applying GENE_ARCH: An evolution-based generative design system. *Advanced Engineering Informatics*, *22*(1), 59–70. http://doi.org/10.1016/j.aei.2007.08.012

Chan, A. S. L. (1960). *The Design of Michell Optimum Structures*. *The college of aeronautics* (Vol. 142). Cranfield.

Clarke, J. a. (2001). *Energy simulation in building design* (2nd ed.). Butterworth-Heinemann.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182–197. http://doi.org/10.1109/4235.996017

Emmerich, M., Grötzner, M., & Schütz, M. (2001). Design of Graph-Based Evolutionary Algorithms: A Case Study for Chemical Process Networks. *Evolutionary Computation*, *9*(3), 329–354. http://doi.org/10.1162/106365601750406028

Emmerich, M. T. M., Hopfe, C. J., Marijt, R., Hensen, J. L. M., Struck, C., & Stoelinga, P. A. L. (2008). Evaluating optimization methodologies for future integration in building performance tools. In *Proceedings of the 8th international conference on Adaptive Computing in Design and Manufacture (ACDM)* (Vol. 1, pp. 1–7).

Fong, K. F., Hanby, V. I., & Chow, T. T. (2006). HVAC system optimization for energy management by evolutionary programming. *Energy and Buildings*, *38*(3), 220–231. http://doi.org/10.1016/j.enbuild.2005.05.008

Hammer, V. B., & Olhoff, N. (2000). Topology optimization of continuum structures subjected to pressure loading. *Structural and Multidisciplinary Optimization*, *19*, 85–92. http://doi.org/10.1007/s001580050088

Hensen, J. L. M. (n.d.). Towards more effective use of building performance simulation in design. In *Proceedings of the 7th international Conference on Design & decision support systems in architecture and urban planning* (pp. 1–16). Eindhoven, the Netherlands.

Hensen, J. L. M., & Lamberts, R. (2011). *Building Performance Simulation for Design and Operation*. (J. L. M. Hensen & L. Lamberto, Eds.). Spon Press.

Hofmeyer, H., & Davila Delgado, J. M. (2015). Co-evolutionary and Genetic Algorithm Based

Building Spatial and Structural Design. *AIEDAM - Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 29*, 351–370. http://doi.org/10.1017/S0890060415000384

Iuspa, L., & Ruocco, E. (2008). Optimum topological design of simply supported composite stiffened panels via genetic algorithms. *Computers and Structures*, *86*, 1718–1737. http://doi.org/10.1016/j.compstruc.2008.02.001

Jang, G. W., Yoon, M. S., & Park, J. H. (2010). Lightweight flatbed trailer design by using topology and thickness optimization. *Structural and Multidisciplinary Optimization*, *41*, 295–307. http://doi.org/10.1007/s00158-009-0409-x

Kang, J. H., & Kim, C. G. (2005). Minimum-weight design of compressively loaded composite plates and stiffened panels for postbuckling strength by Genetic Algorithm. *Composite Structures*, *69*, 239–246. http://doi.org/10.1016/j.compstruct.2004.07.001

Kicinger, R., Arciszewski, T., & De Jong, K. (2005). Evolutionary computation and structural design: A survey of the state-of-the-art. *Computers and Structures*, *83*(23-24), 1943–1978. http://doi.org/10.1016/j.compstruc.2005.03.002

Kramer, R., Schijndel, J. Van, & Schellen, H. (2012). Simplified thermal and hygric building models: A literature review. *Frontiers of Architectural Research*, *1*(4), 318–325. http://doi.org/10.1016/j.foar.2012.09.001

Kramer, R., van Schijndel, J., & Schellen, H. (2013). Inverse modeling of simplified hygrothermal building models to predict and characterize indoor climates. *Building and Environment*, *68*, 87–99. http://doi.org/10.1016/j.buildenv.2013.06.001

Liang, Qing Quan; Xie, Yi Min; Steven, G. P. (2000). Optimal topology design of bracing systems for multistory steel frames. *Structural Engineering*, *126*(7), 823–829.

Marks, W. (1997). Multicriteria optimisation of shape of energy-saving buildings. *Building and Environment*, *32*(4), 331–339. http://doi.org/10.1016/S0360-1323(96)00065-0

Tuhus-Dubrow, D., & Krarti, M. (2010). Genetic-algorithm based approach to optimize building envelope design for residential buildings. *Building and Environment*, *45*(7), 1574–1581. http://doi.org/10.1016/j.buildenv.2010.01.005

Underwood, C. P., & Yik, F. W. H. (2004). *Modelling Methods for Energy in Buildings* (1st ed.). Blackwell Publishing Ltd.

van Schijndel, J., & Kramer, R. (2014). Combining Three Main Modeling Methodologies for Building Physics. In *Proceedings of the 10th Nordic Symposium on Building Physics (NSB 2014). Lund, Sweden* (pp. 558–565). Retrieved from http://www.nsb2014.se/wordpress/wp-content/uploads/2014/07/Complete_fullpapers.pdf

Voll, P., Lampe, M., Wrobel, G., & Bardow, A. (2012). Superstructure-free synthesis and optimization of distributed industrial energy supply systems. *Energy*, *45*, 424–435. http://doi.org/10.1016/j.energy.2012.01.041

Wright, J. A., Loosemore, H. A., & Farmani, R. (2002). Optimization of building thermal design and control by multi-criterion genetic algorithm. *Energy and Buildings*, *34*(9), 959–972. http://doi.org/10.1016/S0378-7788(02)00071-3

Yi, Y. K., & Malkawi, A. M. (2009). Optimizing building form for energy performance based on hierarchical geometry relation. *Automation in Construction*, *18*(6), 825–833. http://doi.org/10.1016/j.autcon.2009.03.006

Annexes

Annex 1	C++ code for the conversion between the Super Cube and Movable Sizable representations	Bound in at the back of the report
Annex 2	C++ code of the visualisation of the MovableSizable-class	Bound in at the back of the report
Annex 3	C++ code of the visualisation of the SuperCube-class	Bound in at the back of the report
Annex 4	C++ code of the classes in the building physics simulation program	Bound in at the back of the report
Annex 5	Matlab code of the state space wall example	Bound in at the back of the report
Annex 6	Input file of the resistance between two states example	Bound in at the back of the report
Annex 7	Input file concrete box example	Bound in at the back of the report
Annex 8	C++ code of the classes in the conformation program	Bound in at the back of the report

Annex 1

C++ code for the conversion between the Super Cube and Movable Sizable representations

```
#ifndef MOVABLESIZABLE_H
#define MOVABLESIZABLE_H
#include "Supercube.h" // for operator overloading, this
                       // also loads in <string> and <vector>

struct RoomMS
{
    int ID;
    double width, depth, height;
    double x, y, z;
};

class MovableSizable
{
private:
    std::vector<RoomMS> rooms;
protected:

public:
    void read_file(std::string);
    void write_file(std::string);
    operator Supercube() const;
    int obtain_room_count();
    RoomMS obtain_room(int room_index);
    bool check_cell(Supercube S, int cell_index, int room_ID,
                    double x_origin, double y_origin,
                    double z_origin) const;

    MovableSizable(Supercube);
    MovableSizable(std::string);
    ~MovableSizable();
};
```

```cpp
#include "MovableSizable.h"
#include "StringTokenizer.h"
#include <fstream> // for if- and ofstream
#include <algorithm> // for vector::sort()
#include <cmath>


void MovableSizable::read_file(std::string filename)
{
    std::ifstream input(filename.c_str());

    while (!input.eof())
    {
        std::string line;
        getline(input,line);
        RoomMS temp_room;

        StringTokenizer strtok = StringTokenizer(line,",");

        std::string t;
        t = strtok.nextToken(); // this is 'R', so skip

        if (t != "R") {}
        else
        {
            t = strtok.nextToken(); // this is 'ID of room'
            temp_room.ID = atoi(t.c_str());

            t = strtok.nextToken(); // this is 'width'
            temp_room.width = atof(t.c_str());

            t = strtok.nextToken(); // this is 'depth'
            temp_room.depth = atof(t.c_str());

            t = strtok.nextToken(); // this is 'height'
            temp_room.height = atof(t.c_str());

            t = strtok.nextToken(); // this is 'x-coordinate'
            temp_room.x = atof(t.c_str());

            t = strtok.nextToken(); // this is 'y-coordinate'
            temp_room.y = atof(t.c_str());

            t = strtok.nextToken(); // this is 'z-coordinate'
            temp_room.z = atof(t.c_str());

            rooms.push_back(temp_room);
        }
    }
}

void MovableSizable::write_file(std::string filename)
{
    std::ofstream output;
    output.open(filename.c_str());

    for (unsigned int i = 0; i < rooms.size(); i++)
    {
        output << "R," << rooms[i].ID << "," << rooms[i].width << "," << rooms[i].depth << ","
               << rooms[i].height << "," << rooms[i].x << "," << rooms[i].y << "," <<
               rooms[i].z << std::endl;
    }
```

```
    output.close();
}

int MovableSizable::obtain_room_count()
{
    return rooms.size();
}


RoomMS MovableSizable::obtain_room(int room_index)
{
    return rooms[room_index];
}


MovableSizable::operator Supercube() const // conversion from MovableSizable to Supercube
{
    Supercube S;
    std::vector<double> x_values, y_values, z_values; // these vectors will contain all x-
    y- and z-values of the MS representation

    for (unsigned int i = 0; i < rooms.size(); i++) // this stacks every x- y- and
    z-coordinate of every room in the respective vectors
    {
        x_values.push_back(rooms[i].x);
        x_values.push_back(rooms[i].x+rooms[i].width);
        y_values.push_back(rooms[i].y);
        y_values.push_back(rooms[i].y+rooms[i].depth);
        z_values.push_back(rooms[i].z);
        z_values.push_back(rooms[i].z+rooms[i].height);
    }

    sort(x_values.begin(), x_values.end()); // sorts all values in the vectors in ascending
    order
    sort(y_values.begin(), y_values.end());
    sort(z_values.begin(), z_values.end());
    x_values.erase(unique(x_values.begin(), x_values.end()), x_values.end()); // erases all
    duplicates from the vectors
    y_values.erase(unique(y_values.begin(), y_values.end()), y_values.end());
    z_values.erase(unique(z_values.begin(), z_values.end()), z_values.end());

    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

    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
        { S.stack_w_value(x_values[i+1] - x_values[i]); }
    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
        { S.stack_d_value(y_values[i+1] - y_values[i]); }
    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
        { S.stack_h_value(z_values[i+1] - z_values[i]); }

    int cube_size = S.w_size()*S.d_size()*S.h_size(); // initialize super cube size
    std::vector<int> b_values_row; // initializes a vector for the b_values matrix

    for (unsigned int room_index = 0; room_index < rooms.size(); room_index++)  // computes
    a row of the b_values matrix and adds these to the matrix for each room
    {
        b_values_row.clear();
```

```
        b_values_row.push_back(room_index + 1); // index 0 contains the room ID, for the
        super cube the count starts again from 1.

        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
        {
            b_values_row.push_back((check_cell(S, cell_index, room_index, x_origin,
            y_origin, z_origin) ? 1 : 0)); // If a cell belongs to the room assign 1 to
            index, if not 0
        }

        S.stack_b_value_row(b_values_row); // adds the row to the b_values matrix
    }

    return S;
}

bool MovableSizable::check_cell(Supercube S, int cell_index, int room_index, double
x_origin, double y_origin, double z_origin) const
{
    int w_index = S.get_w_index(cell_index);
    int d_index = S.get_d_index(cell_index);
    int h_index = S.get_h_index(cell_index);
    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

    for (int k = 0; k < w_index; k++)    // compute the x_coordinates of the cell's origin
        { x_coor += S.request_w(k); }
    for (int k = 0; k < d_index; k++)    // compute the y_coordinates of the cell's origin
        { y_coor += S.request_d(k); }
    for (int k = 0; k < h_index; k++)    // compute the z_coordinates of the cell's origin
        { z_coor += S.request_h(k); }

    if((x_coor >= rooms[room_index].x && x_coor < rooms[room_index].x +
    rooms[room_index].width) &&
       (y_coor >= rooms[room_index].y && y_coor < rooms[room_index].y +
       rooms[room_index].depth) &&
       (z_coor >= rooms[room_index].z && z_coor < rooms[room_index].z +
       rooms[room_index].height))  // if a cell is within the bounds of room r return true
        { return true; }
    else
        { return false;}
}

MovableSizable::MovableSizable(Supercube S) //conversion from Supercube to MovableSizable
{
    int w_origin = S.w_size();
    int d_origin = S.d_size();
    int h_origin = S.h_size();

    for (unsigned int cell = 1; cell <= (S.w_size()*S.d_size()*S.h_size()); cell++) //
    starts with 1, since first index is the room ID // each cell is checked for each room,
    whether it describes a room
    {
        int w_index = S.get_w_index(cell);
        int d_index = S.get_d_index(cell);
        int h_index = S.get_h_index(cell);  // compute the grid indexes of the cell

        for (unsigned int room = 0; room < S.b_size(); room++) // check each room for the
        considered cell, whether it describes a room
        {
            if (S.request_b(room, cell) == 1) // if it does describe a room, then update the
```

```
            origin indexes
                {
                    if (w_index < w_origin) { w_origin = w_index; }
                    if (d_index < d_origin) { d_origin = d_index; }
                    if (h_index < h_origin) { h_origin = h_index; }
                }
        }
    }


    for (unsigned int room_ID = 0; room_ID < S.b_size(); room_ID++)
    {
        RoomMS room = {0, 0.0 ,0.0 ,0.0 ,0.0 ,0.0 ,0.0}; // initializing all values in an
        object of the RoomMS structure
        room.ID = room_ID; // assigning the room ID to the RoomMS object
        int maximum = 0, minimum = 0;   // initializing minimum and maximum indexes of the
        space with id: i+1
        for (unsigned int cell = 0; cell < S.b_row_size(room_ID); cell++)   // finds the min
        and max indexes
        {
            if (minimum == 0 && S.request_b(room_ID, cell) == 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
                {minimum = cell;}
            if (S.request_b(room_ID, cell) == 1)    // finds the last index with value 1 in
            the cell_array, this is the room's outmost cell assuming spaces are cuboid
                {maximum = cell;}
        }

        int min_w = S.get_w_index(minimum), max_w = S.get_w_index(maximum),
            min_d = S.get_d_index(minimum), max_d = S.get_d_index(maximum),
            min_h = S.get_h_index(minimum), max_h = S.get_h_index(maximum);   // computes
            the grid indexes of the min and max cells

        for (int i = w_origin; i <= max_w; i++) // moves through each w_index in which cells
        are assigned to the considered room
        {
            if (i < min_w) { room.x += S.request_w(i); }
            if (i >= min_w) { room.width += S.request_w(i); }
        }

        for (int i = d_origin; i <= max_d; i++) // moves through each d_index in which cells
        are assigned to the considered room
        {
            if (i < min_d) { room.y += S.request_d(i); }
            if (i >= min_d) { room.depth += S.request_d(i); }
        }

        for (int i = h_origin; i <= max_h; i++) // moves through each h_index in which cells
        are assigned to the considered room
        {
            if (i < min_h) { room.z += S.request_h(i); }
            if (i >= min_h) { room.height += S.request_h(i); }
        }

        rooms.push_back(room);
    }
}

MovableSizable::MovableSizable(std::string filename)
{
    read_file(filename);//ctor
```

```
}

MovableSizable::~MovableSizable()
{
    //dtor
}
```

```
#ifndef SUPERCUBE_H
#define SUPERCUBE_H
#include <vector>
#include <string>


class Supercube
{
private:
    std::vector<double> w_values;
    std::vector<double> d_values;
    std::vector<double> h_values;
    std::vector<std::vector<int>> b_values;
protected:

public:
    void read_file(std::string);
    void write_file(std::string);
    int get_w_index(int cell_index);
    int get_d_index(int cell_index);
    int get_h_index(int cell_index);

    unsigned int w_size();
    unsigned int d_size();
    unsigned int h_size();
    unsigned int b_size();
    unsigned int b_row_size(int room_ID);

    double request_w(int index);
    double request_d(int index);
    double request_h(int index);
    int request_b(int room_ID, int cell_index);

    void stack_w_value(double w_value);
    void stack_d_value(double d_value);
    void stack_h_value(double h_value);
    void stack_b_value_row(std::vector<int> b_value_row);


    Supercube(std::string filename);
    Supercube();
    ~Supercube();

};

#endif // SUPERCUBE_H
```

```
#include "Supercube.h"
#include "StringTokenizer.h"
#include <fstream>

void Supercube::read_file(std::string filename)
{
    std::ifstream input(filename.c_str());

    while (!input.eof())
    {
        std::string line;
        getline(input,line);

        StringTokenizer strtok = StringTokenizer(line,",");
        std::string t;
        t = strtok.nextToken();

        if (t == "b")
        {
            t = strtok.nextToken();
            int ID = atoi(t.c_str());
            std::vector<int> row; row.push_back(ID);
            b_values.push_back(row);
            while (strtok.hasMoreTokens())
            {
                t = strtok.nextToken();
                b_values[ID-1].push_back(atoi(t.c_str()));
            }
        }
        if (t == "w")
        {
            while (strtok.hasMoreTokens())
            {
                t = strtok.nextToken();
                w_values.push_back(atof(t.c_str()));
            }
        }
        if (t == "d")
        {
            while (strtok.hasMoreTokens())
            {
                t = strtok.nextToken();
                d_values.push_back(atof(t.c_str()));
            }
        }
        if (t == "h")
        {
            while (strtok.hasMoreTokens())
            {
                t = strtok.nextToken();
                h_values.push_back(atof(t.c_str()));
            }
        }
    } // add check to see if all b_value strings are complete, i.e. they all contain
    w_values.size()*d_values.size()*h_values.size() = b_values[i].size() +1 (first entry =
    ID) for each i?
}

unsigned int Supercube::w_size()
{
    return w_values.size();
}
```

```
unsigned int Supercube::d_size()
{
    return d_values.size();
}
unsigned int Supercube::h_size()
{
    return h_values.size();
}
unsigned int Supercube::b_size()
{
    return b_values.size();
}
unsigned int Supercube::b_row_size(int room_ID)
{
    return b_values[room_ID].size();
}
double Supercube::request_w(int index)
{
    return w_values[index];
}
double Supercube::request_d(int index)
{
    return d_values[index];
}
double Supercube::request_h(int index)
{
    return h_values[index];
}
int Supercube::request_b(int room_ID, int cell_index)
{
    return b_values[room_ID][cell_index];
}

void Supercube::stack_w_value(double w_value)
{
    w_values.push_back(w_value);
}
void Supercube::stack_d_value(double d_value)
{
    d_values.push_back(d_value);
}
void Supercube::stack_h_value(double h_value)
{
    h_values.push_back(h_value);
}
void Supercube::stack_b_value_row(std::vector<int> b_value_row)
{
    b_values.push_back(b_value_row);
}

void Supercube::write_file(std::string filename)
{
    std::ofstream output;
    output.open(filename.c_str());

    output << "Super cube design space\n" << std::endl;
    output << "Vector w:\nw,";

    for (unsigned int i = 0; i < w_values.size(); i++)
    {
        output << w_values[i];
        if (i < w_values.size() -1) output << ",";
```

```
    }

    output << "\nVector d:\nd,";

    for (unsigned int i = 0; i < d_values.size(); i++)
    {
        output << d_values[i];
        if (i < d_values.size() -1) output << ",";
    }

    output << "\nVector h:\nh,";

    for (unsigned int i = 0; i < h_values.size(); i++)
    {
        output << h_values[i];
        if (i < h_values.size() -1) output << ",";
    }

    output << "\nVector b:" << std::endl;

    for (unsigned int i = 0; i < b_values.size(); i++)
    {
        output << "b," << b_values[i][0] << ",";

        for(unsigned int j = 1; j < b_values[i].size(); j++)
        {
            output << b_values[i][j];
            if (j < b_values[i].size() -1) output << ", ";
        }
        output << std::endl;
    }

    output.close();
}

int Supercube::get_w_index(int cell_index)
{
    return ((cell_index-1)/(h_values.size()*d_values.size()));
}

int Supercube::get_d_index(int cell_index)
{
    return (((cell_index-1) -
    get_w_index(cell_index)*(h_values.size()*d_values.size()))/(h_values.size()));
}

int Supercube::get_h_index(int cell_index)
{
    return (cell_index-1) - get_w_index(cell_index)*(h_values.size()*d_values.size()) -
    get_d_index(cell_index)*h_values.size();
}

Supercube::Supercube(std::string filename)
{
    read_file(filename); //ctor
}

Supercube::Supercube()
{

}
```

```
Supercube::~Supercube()
{
    //dtor
}
```

Annex 2

C++ code of the visualisation of the MovableSizable-class

```
#ifndef MOVABLESIZABLE_MODEL_H

#define MOVABLESIZABLE_MODEL_H

#include "model.h"
#include "MovableSizable.h"


class MovableSizable_model : public model
{
public:
    MovableSizable_model(MovableSizable);
    ~MovableSizable_model();
    void render(const camera &cam) const;
    const string get_description();
private:
    list<polygon*> polygons;
    list<label*>   labels;
    polygon_props  pprops;
    line_props     lprops;
    label_props    lbprops;
    random_bsp     *pbsp;
};

#endif // MOVABLESIZABLE_MODEL_H
```

```cpp
#include "MovableSizable_model.h"
#include "bsp.h"

// hh, below is the definition of the constructor of setofrooms_model as defined in
setofrooms_model.h
// hh, see datatypes.h for usage of pprops:
// hh, rgba ambient,diffuse,specular, emission;
// hh, float shininess;
// hh, bool translucent, wosided;


MovableSizable_model::MovableSizable_model(MovableSizable MS)
    /*
  : pprops( rgba(0.0f, 0.0f, 0.0f, 0.0f), //ambient
            rgba(8.0f, 0.0f, 0.0f, 0.1f), //diffuse
            rgba(0.0f, 0.0f, 0.0f, 0.0f), //specular
            rgba(0.0f, 0.0f, 0.0f, 0.0f), //emission
            60.0f, true, true)
     */

  : pprops( rgba(0.4f, 0.1f, 0.1f, 0.3f), //ambient
            rgba(0.8f, 0.2f, 0.2f, 0.3f), //diffuse
            rgba(1.0f, 0.2f, 0.2f, 0.3f), //specular
            rgba(0.0f, 0.0f, 0.0f, 0.0f), //emission
            60.0f, true, true)


{
    vertex max, min;

    for(int room_index = 0; room_index < MS.obtain_room_count(); room_index++)
    {
        RoomMS temp = MS.obtain_room(room_index);

        min = vect3d(temp.x,temp.z,-(temp.y+temp.depth));
        max = vect3d(temp.x+temp.width,temp.z+temp.height,-temp.y);

        add_cube(&pprops, &lprops, min, max, polygons);
        ostringstream out; out << temp.ID; string ID = out.str(); // cast the int value of
        ID as a string
        labels.push_back(create_label(&lbprops, ID,
                        min + ((max-min)/2.0)));
    }
    pbsp = new random_bsp(polygons);
}

MovableSizable_model::~MovableSizable_model()
{
    delete pbsp;

    for (list<polygon*>::iterator pit = polygons.begin();
         pit != polygons.end(); pit++)
        delete *pit;

    for (list<label*>::iterator lbit = labels.begin();
         lbit != labels.end(); lbit++)
        delete *lbit;
}
```

Annex 3

C++ code of the visualisation of the SuperCube-class

```
#ifndef SUPERCUBE_MODEL_H
#define SUPERCUBE_MODEL_H
#include "Supercube.h"
#include "model.h"

class Supercube_model : public model
{
public:
    Supercube_model(Supercube);
    ~Supercube_model();
    void render(const camera &cam) const;
    const string get_description();

protected:

private:
    list<polygon*> polygons;
    list<label*>   labels;

    polygon_props  pprops_empty_cell;
    polygon_props  pprops_active_cell;

    line_props     lprops;
    label_props    lbprops;
    random_bsp     *pbsp;
};

#endif // SUPERCUBE_MODEL_H
```

```
#include "Supercube_model.h"
#include "bsp.h"

// hh, below is the definition of the constructor of setofrooms_model as defined in
setofrooms_model.h
// hh, see datatypes.h for usage of pprops:
// hh, rgba ambient,diffuse,specular, emission;
// hh, float shininess;
// hh, bool translucent, wosided;
Supercube_model::Supercube_model(Supercube S)

    /*
  : pprops( rgba(0.0f, 0.0f, 0.0f, 0.0f), //ambient
            rgba(8.0f, 0.0f, 0.0f, 0.1f), //diffuse
            rgba(0.0f, 0.0f, 0.0f, 0.0f), //specular
            rgba(0.0f, 0.0f, 0.0f, 0.0f), //emission
            60.0f, true, true)
     */

  : pprops( rgba(0.4f, 0.1f, 0.1f, 0.3f), //ambient
            rgba(0.8f, 0.2f, 0.2f, 0.3f), //diffuse
            rgba(1.0f, 0.2f, 0.2f, 0.3f), //specular
            rgba(0.0f, 0.0f, 0.0f, 0.0f), //emission
            60.0f, true, true)


{
    vertex max, min;

    /*for(int room_index = 0; room_index < MS.obtain_room_count(); room_index++)
    {
        RoomMS temp = MS.obtain_room(room_index);

        min = vect3d(temp.x,temp.z,-(temp.y+temp.depth));
        max = vect3d(temp.x+temp.width,temp.z+temp.height,-temp.y);

        add_cube(&pprops, &lprops, min, max, polygons);
        ostringstream out; out << temp.ID; string ID = out.str(); // cast the int value of
        ID as a string
        labels.push_back(create_label(&lbprops, ID,
                         min + ((max-min)/2.0)));
    }*/
    pbsp = new random_bsp(polygons);
}

Supercube_model::~Supercube_model()
{
    delete pbsp;

    for (list<polygon*>::iterator pit = polygons.begin();
         pit != polygons.end(); pit++)
        delete *pit;

    for (list<label*>::iterator lbit = labels.begin();
         lbit != labels.end(); lbit++)
        delete *lbit;
}
```

Annex 4

C++ code of the classes in the building physics simulation program

```
#ifndef BP_STATE_H
#define BP_STATE_H
#include <Eigen/Core>
#include <iostream>
#include <iomanip>

struct Adj_State;

class BP_State
{
private:

public:
    BP_State();
    virtual ~BP_State();

    virtual bool is_dep();
    virtual bool is_indep();
    virtual bool is_space();
    virtual bool is_weather_profile();
    virtual bool is_ground_profile();

    virtual unsigned int get_index() = 0;
    //virtual void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u) = 0; // is this required here?
    virtual void add_adj_state(Adj_State new_adj_state) = 0; // required in Dep-State, but
    not Indep_state, however it should be possible to call this function on base class State
    (This fucntion will remain empty in class Indep_state)
};

struct Adj_State
{
    BP_State* m_state_ptr; // pointer to adjacent state
    double m_resistance; // flux resistance to that state
};

#endif // BP_STATE_H
```

```cpp
#include "BP_State.h"

BP_State::BP_State()
{
   //ctor
}

BP_State::~BP_State()
{
    //dtor
}

bool BP_State::is_dep()
{
    return false;
}

bool BP_State::is_indep()
{
    return false;
}

bool BP_State::is_space()
{
    return false;
}

bool BP_State::is_weather_profile()
{
    return false;
}

bool BP_State::is_ground_profile()
{
    return false;
}
```

```
#ifndef BP_INDEP_STATE_H
#define BP_INDEP_STATE_H

#include "BP_State.h"

class BP_Indep_State : public BP_State // this class will read temperature profiles like
weather files or ground temperature etc.
{
protected:
    static unsigned int m_count;
    unsigned int m_index;
public:
    BP_Indep_State();
    ~BP_Indep_State();

    virtual unsigned int get_index(); // for testing purposes
    virtual unsigned int get_count();

    virtual bool is_indep();
    virtual void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u);
    virtual double get_temp();
    virtual void add_adj_state(Adj_State new_adj_state);
};

#endif // BP_INDEP_STATE_H
```

```cpp
#include "BP_Indep_State.h"

unsigned int BP_Indep_State::m_count = 1; // first count goes to 1 for constants (like
heating and cooling P-switch)

BP_Indep_State::BP_Indep_State()
{
    m_index = m_count++;
    //ctor
}

BP_Indep_State::~BP_Indep_State()
{
    //dtor
}

unsigned int BP_Indep_State::get_index() // for testing purposes
{
    return m_index;
}

unsigned int BP_Indep_State::get_count()
{
    return m_count;
}

void BP_Indep_State::update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
Eigen::VectorXd& u)
{

}

double BP_Indep_State::get_temp()
{
    // empty will be defined in derived classes
}

void BP_Indep_State::add_adj_state(Adj_State new_adj_state)
{
    // empty as it will be called upon for independant states however, it should not add
    adjacent states to the independant class as they have no influence on the state.
}

bool BP_Indep_State::is_indep()
{
    return true;
}
```

```
#ifndef BP_DEP_STATE_H
#define BP_DEP_STATE_H

#include "BP_State.h"
#include <vector>
#include <Eigen/Core>

class BP_Dep_State : public BP_State // dependant state, state is dependant of system.
{
protected:
    std::vector<Adj_State> m_connections; // this vector holds structures of adjacent states
    and the resistance to flux to these states
    double m_capacitance; // capacitance of this state
    unsigned int m_dep_index;
    static unsigned int m_count;
public:
    BP_Dep_State(); //ctor
    virtual ~BP_Dep_State(); //dtor

    virtual unsigned int get_index(); // for testing purposes
    virtual unsigned int get_count();
    virtual double get_capa(); // for testing purposes
    virtual void get_res(); // for testing purposes

    virtual bool is_dep();
    virtual void init_sys(Eigen::MatrixXd& A, Eigen::MatrixXd& B); // updates the invariant
    fluxes in the A and B matrices of the state space system
    virtual void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u, double time_increment) = 0;  // updates the variant fluxes in the A
    and B matrices of the state space system
    virtual void add_adj_state(Adj_State new_adj_state); // adds a state (with respective
    resistance) to the vector m_capacitors

};

#endif // BP_DEP_STATE_H
```

```cpp
#include "BP_Dep_State.h"
//#include <BP_Indep_State.h>
#include <cstdlib>

unsigned int BP_Dep_State::m_count = 0;

BP_Dep_State::BP_Dep_State() : BP_State()
{
    m_dep_index = m_count++;
    // ctor
}

BP_Dep_State::~BP_Dep_State()
{
    //dtor
}

unsigned int BP_Dep_State::get_index() // for testing purposes
{
    return m_dep_index;
}

unsigned int BP_Dep_State::get_count()
{
    return m_count;
}

double BP_Dep_State::get_capa() // for testing purposes
{
    return m_capacitance;
}

void BP_Dep_State::get_res() // for testing purposes
{
    for (unsigned int i = 0; i < m_connections.size(); i++)
    {
        std::cout << m_connections[i].m_resistance << std::endl;
    }
}

bool BP_Dep_State::is_dep()
{
    return true;
}

void BP_Dep_State::init_sys(Eigen::MatrixXd& A, Eigen::MatrixXd& B)
{
    for (unsigned int i = 0; i < m_connections.size(); i++)
    {
        if (m_connections[i].m_state_ptr->is_dep())
        {
            A(this->get_index(), this->get_index()) += -1/(m_capacitance*
            m_connections[i].m_resistance);
            A(this->get_index(), m_connections[i].m_state_ptr->get_index()) +=
            1/(m_capacitance* m_connections[i].m_resistance);
        }
        else if (m_connections[i].m_state_ptr->is_indep())
        {
            A(this->get_index(), this->get_index()) += -1/(m_capacitance*
            m_connections[i].m_resistance);
            B(this->get_index(), m_connections[i].m_state_ptr->get_index()) +=
            1/(m_capacitance* m_connections[i].m_resistance);
```

```
        }
        else
        {
            std::cout << "Error assembling system, exiting... " << std::endl;
            exit(1);
        }
    }
}

void BP_Dep_State::add_adj_state(Adj_State new_adj_state)// adds a state (with respective
resistance) to the vector m_capacitors
{
    m_connections.push_back(new_adj_state);
}
```

```
#ifndef BP_CONSTRUCTION_H
#define BP_CONSTRUCTION_H

#include <string>
#include <vector>

struct BP_Material // holds the name and properties of a material
{
    std::string m_material_ID;
    std::string m_name; // name/description of the material
    double m_spec_weight, m_spec_heat, m_therm_conductivity; // material properties
};

struct BP_Con_Layer // holds the material and thickness of a construction layer
{
    BP_Material m_material; // material type
    double m_thickness; // layer thickness
};

struct BP_Construction // don't forget air resistances
{
    std::string m_construction_ID;
    std::vector<BP_Con_Layer> m_layers; // vector lists layers from side 1 to side 2, side 1
    is default outside.
    double m_total_thickness, m_capacitance_per_area; // total wall thickness and total
    capacity of the entire wall
    double m_resistance_to_side_1, m_resistance_to_side_2; // resistances per square meter
    between this state and adjacent states
    double m_measure_point; // fraction of wall thickness whereat the temperature of the
    wall is measured, measuring from side 1

    // functions
    double get_capacitance_per_area(); // returns the heat capacity per square metre of this
    construction's surface
    double get_resistance_to_side_1(); // returns the resistance of this construction to
    flux to side one
    double get_resistance_to_side_2(); // returns the resistance of this construction to
    flux to side two
};

#endif // BP_CONSTRUCTION_H
```

```cpp
#include <BP_Construction.h>

double BP_Construction::get_capacitance_per_area() // getter function for construction's
capacity per square metre
{
    return m_capacitance_per_area;
}

double BP_Construction::get_resistance_to_side_1() // getter function for construction's
resistance to flux to side one
{
    return m_resistance_to_side_1;
}

double BP_Construction::get_resistance_to_side_2() // getter function for construction's
resistance to flux to side two
{
    return m_resistance_to_side_2;
}
```

```
#ifndef BP_WALL_H
#define BP_WALL_H

#include "BP_Dep_State.h"
#include "BP_Construction.h"


class BP_Wall : public BP_Dep_State
{
private:
    BP_Construction* m_construction; // this is the construction of which the wall is made
    double m_area; // surface area of the wall
    std::string m_ID;
    BP_State* m_side_1; // the temperature state at side one of the wall
    BP_State* m_side_2; // the temperature state at side two of the wall
public:
    BP_Wall(std::string wall_ID, const double area, BP_Construction* construction,
    BP_State*& side_1, BP_State*& side_2); // ctor
    ~BP_Wall(); // dtor

    std::string get_ID();
    void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u, double time_increment); // updates the A and B matrices of the state
    space system
};

#endif // BP_WALL_H
```

```cpp
#include "BP_Wall.h"
#include <cstdlib>

BP_Wall::BP_Wall(std::string wall_ID, const double area, BP_Construction* construction,
BP_State*& side_1, BP_State*& side_2) : BP_Dep_State()
{
    m_ID = wall_ID;
    m_area = area; // area in square meters
    m_capacitance = (construction->get_capacitance_per_area()) * (area); // capacitance in J/K
    m_side_1 = side_1; // temperature state at side one of the wall
    m_side_2 = side_2; // temperature state at side two of the wall

    double trans_res_side_1, trans_res_side_2;
    if (m_side_1->is_space())
    {
        trans_res_side_1 = 0.13;
    }
    else if (m_side_1->is_weather_profile())
    {
        trans_res_side_1 = 0.04;
    }
    else if (m_side_1->is_ground_profile())
    {
        trans_res_side_1 = 0.0;
    }
    else
    {
        std::cout << "Something went wrong in assigning the transition resistances,
        exiting.. " << std::endl;
        exit(1);
    }

    if (m_side_2->is_space())
    {
        trans_res_side_2 = 0.13;
    }
    else if (m_side_2->is_weather_profile())
    {
        trans_res_side_2 = 0.04;
    }
    else if (m_side_2->is_ground_profile())
    {
        trans_res_side_2 = 0.0;
    }
    else
    {
        std::cout << "Something went wrong in assigning the transition resistances,
        exiting.. " << std::endl;
        exit(1);
    }

    // Create an Adjacent State structure for adding information to this and adjacent states
    (structure is declared in BP_State.h)
    Adj_State adj_state;

    // Add adjacent states to this state
    adj_state.m_state_ptr = this;
    adj_state.m_resistance = ((construction->get_resistance_to_side_1() + trans_res_side_1)
    / m_area);
    m_side_1->add_adj_state(adj_state);

    adj_state.m_resistance = ((construction->get_resistance_to_side_2() + trans_res_side_2)
```

```
    / m_area);
    m_side_2->add_adj_state(adj_state);

    // Add this state as adjacent state to adjacent states
    adj_state.m_state_ptr = m_side_1;
    adj_state.m_resistance = ((construction->get_resistance_to_side_1() + trans_res_side_1)
    / m_area);
    this->add_adj_state(adj_state);

    adj_state.m_state_ptr = m_side_2;
    adj_state.m_resistance = ((construction->get_resistance_to_side_2() + trans_res_side_2)
    / m_area);
    this->add_adj_state(adj_state);
    //ctor
}

BP_Wall::~BP_Wall()
{
    //dtor
}


void BP_Wall::update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
Eigen::VectorXd& u, double time_increment) // updates the A and B matrices of the state
space system
{

}

std::string BP_Wall::get_ID()
{
    return m_ID;
}
```

```
#ifndef BP_FLOOR_H
#define BP_FLOOR_H

#include <BP_Dep_State.h>
#include "BP_Construction.h"


class BP_Floor : public BP_Dep_State
{
private:
    BP_Construction* m_construction; // this is the construction of which the floor is made
    double m_area; // surface area of the floor
    std::string m_ID; // floor ID
    BP_State* m_side_1; // the temperature state at side one of the floor
    BP_State* m_side_2; // the temperature state at side two of the floor
public:
    BP_Floor(std::string floor_ID, const double area, BP_Construction* construction,
    BP_State*& side_1, BP_State*& side_2);
    ~BP_Floor();

    std::string get_ID();
    void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u, double time_increment); // updates the A and B matrices of the state
    space system
};

#endif // BP_FLOOR_H
```

```cpp
#include "BP_Floor.h"

BP_Floor::BP_Floor(std::string floor_ID, const double area, BP_Construction* construction,
BP_State*& side_1, BP_State*& side_2)
{
    m_ID = floor_ID;
    m_area = area; // area in square meters
    m_capacitance = (construction->get_capacitance_per_area()) * (area); // capacitance in J/K
    m_side_1 = side_1; // temperature state at side one of the floor
    m_side_2 = side_2; // temperature state at side two of the floor

    double trans_res_side_1, trans_res_side_2;
    if (m_side_1->is_space())
    {
        trans_res_side_1 = 0.13;
    }
    else if (m_side_1->is_weather_profile())
    {
        trans_res_side_1 = 0.04;
    }
    else if (m_side_1->is_ground_profile())
    {
        trans_res_side_1 = 0.0;
    }
    else
    {
        std::cout << "Something went wrong in assigning the transition resistances,
        exiting.. " << std::endl;
        exit(1);
    }

    if (m_side_2->is_space())
    {
        trans_res_side_2 = 0.13;
    }
    else if (m_side_2->is_weather_profile())
    {
        trans_res_side_2 = 0.04;
    }
    else if (m_side_2->is_ground_profile())
    {
        trans_res_side_2 = 0.0;
    }
    else
    {
        std::cout << "Something went wrong in assigning the transition resistances,
        exiting.. " << std::endl;
        exit(1);
    }

    // Create an Adjacent State structure for adding information to this and adjacent states
    (structure is declared in BP_State.h)
    Adj_State adj_state;

    // Add adjacent states to this state
    adj_state.m_state_ptr = this;
    adj_state.m_resistance = ((construction->get_resistance_to_side_1() + trans_res_side_1)
    / m_area);
    m_side_1->add_adj_state(adj_state);

    adj_state.m_resistance = ((construction->get_resistance_to_side_2() + trans_res_side_2)
    / m_area);
```

```
    m_side_2->add_adj_state(adj_state);

    // Add this state as adjacent state to adjacent states
    adj_state.m_state_ptr = m_side_1;
    adj_state.m_resistance = ((construction->get_resistance_to_side_1() + trans_res_side_1)
    / m_area);
    this->add_adj_state(adj_state);

    adj_state.m_state_ptr = m_side_2;

    adj_state.m_resistance = ((construction->get_resistance_to_side_2() + trans_res_side_2)
    / m_area);
    this->add_adj_state(adj_state);
    //ctor
}

BP_Floor::~BP_Floor()
{
    //dtor
}

void BP_Floor::update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
Eigen::VectorXd& u, double time_increment) // updates the A and B matrices of the state
space system
{

}

std::string BP_Floor::get_ID()
{
    return m_ID;
}
```

```
#ifndef BP_WINDOW_H
#define BP_WINDOW_H

#include "BP_Dep_State.h"
#include "BP_Construction.h"

class BP_Window : public BP_Dep_State
{
private:
    double m_area; // surface area of the wall
    double m_U_value; // heat resistance of the window
    std::string m_ID;
    BP_State* m_side_1; // the temperature state at side one of the wall
    BP_State* m_side_2; // the temperature state at side two of the wall
public:
    BP_Window(std::string window_ID, const double area, double U_value, double
    capacitance_per_area, BP_State*& side_1, BP_State*& side_2); // ctor
    ~BP_Window(); // dtor

    std::string get_ID();
    void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u, double time_increment); // updates the A and B matrices of the state
    space system
};

#endif // BP_WINDOW_H
```

```cpp
#include "BP_Window.h"
#include <cstdlib>

BP_Window::BP_Window(std::string window_ID, const double area, double U_value, double
capacitance_per_area, BP_State*& side_1, BP_State*& side_2) : BP_Dep_State()
{
    m_ID = window_ID;
    m_U_value = U_value;
    m_area = area; // area in square meters
    m_capacitance = capacitance_per_area * area; // capacitance in J/K
    m_side_1 = side_1; // temperature state at side one of the wall
    m_side_2 = side_2; // temperature state at side two of the wall

    // Create an Adjacent State structure for adding information to this and adjacent states
    (structure is declared in BP_State.h)
    Adj_State adj_state;

    // Add adjacent states to this state
    adj_state.m_state_ptr = this;
    adj_state.m_resistance = (1/(2*m_U_value*m_area));
    m_side_1->add_adj_state(adj_state);

    adj_state.m_resistance = (1/(2*m_U_value*m_area));
    m_side_2->add_adj_state(adj_state);

    // Add this state as adjacent state to adjacent states
    adj_state.m_state_ptr = m_side_1;
    adj_state.m_resistance = (1/(2*m_U_value*m_area));
    this->add_adj_state(adj_state);

    adj_state.m_state_ptr = m_side_2;
    adj_state.m_resistance = (1/(2*m_U_value*m_area));
    this->add_adj_state(adj_state);
    //ctor
}

BP_Window::~BP_Window()
{
    //dtor
}


void BP_Window::update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
Eigen::VectorXd& u, double time_increment) // updates the A and B matrices of the state
space system
{

}

std::string BP_Window::get_ID()
{
    return m_ID;
}
```

```
#ifndef BP_SPACE_H
#define BP_SPACE_H

#include "BP_Dep_State.h"
#include <BP_Weather_Profile.h>

class BP_Space : public BP_Dep_State
{
private:
    double m_volume; // volume of the space
    double m_heat_cap, m_cool_cap;
    double m_heat_set_point, m_cool_sep_point;
    double m_sum_heat_power, m_sum_cool_power;
    double m_ACH;
    std::string m_ID;
public:
    BP_Space(std::string ID, double volume, double heating_capacity, double
    cooling_capacity, double heat_set_point, double cool_set_point, double ACH,
    BP_Weather_Profile* outside_ptr); // initialize building space, with volume and initial
    temperature of that building space
    ~BP_Space();

    bool is_space();

    void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u, double time_increment); // updates the A and B matrices of the state
    space system
    std::string get_ID(); // getter function for Space ID

    void set_power_count_zero();

    void get_power_count()
    {
        std::cout << "Heating: " << m_sum_heat_power << ", Cooling: " << m_sum_cool_power <<
        std::endl;
    }

    void get_setpoints()
    {
        std::cout << "Heating: " << m_heat_set_point << ", Cooling: " << m_cool_sep_point <<
        std::endl;
    }
};

#endif // BP_SPACE_H
```

```
#ifndef BP_SPACE_H
#define BP_SPACE_H

#include "BP_Dep_State.h"
#include <BP_Weather_Profile.h>

class BP_Space : public BP_Dep_State
{
private:
    double m_volume; // volume of the space
    double m_heat_cap, m_cool_cap;
    double m_heat_set_point, m_cool_sep_point;
    double m_sum_heat_power, m_sum_cool_power;
    double m_ACH;
    std::string m_ID;
public:
    BP_Space(std::string ID, double volume, double heating_capacity, double
    cooling_capacity, double heat_set_point, double cool_set_point, double ACH,
    BP_Weather_Profile* outside_ptr); // initialize building space, with volume and initial
    temperature of that building space
    ~BP_Space();

    bool is_space();

    void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u, double time_increment); // updates the A and B matrices of the state
    space system
    std::string get_ID(); // getter function for Space ID

    void set_power_count_zero();

    void get_power_count()
    {
        std::cout << "Heating: " << m_sum_heat_power << ", Cooling: " << m_sum_cool_power <<
        std::endl;
    }

    void get_setpoints()
    {
        std::cout << "Heating: " << m_heat_set_point << ", Cooling: " << m_cool_sep_point <<
        std::endl;
    }
};

#endif // BP_SPACE_H
```

```
#ifndef BP_DATE_TIME_H
#define BP_DATE_TIME_H

#include "BP_Weekdays.h"

#include <string>

class BP_Date_Time
{
private:
    int m_year, m_month, m_day, m_hour;
    BP_Weekdays m_day_name;
public:
    BP_Date_Time(int year, int month, int day, int hour, std::string day_name);
    BP_Date_Time(int year, int month, int day, int hour);
    BP_Date_Time();
    ~BP_Date_Time();

    BP_Weekdays get_day_name();

    int get_day();
    int get_month();
    int get_year();
    int get_hour();

    bool operator == (BP_Date_Time rhs);
    bool operator != (BP_Date_Time rhs);
};

#endif // BP_DATE_TIME_H
```

```cpp
#include "BP_Date_Time.h"

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

BP_Date_Time::BP_Date_Time(int year, int month, int day, int hour, std::string day_name)
{
    m_year = year;
    m_month = month;
    m_day = day;
    m_hour = hour;

    boost::to_upper(day_name);

    if (day_name == "DEFAULT")
    {
        m_day_name = BP_Weekdays(1);
    }
    else
    {
        m_day_name = BP_Weekdays(day_name);
    }

    // this does not work properly yet
    std::tm time_in = { 0, 0, 0, // second, minute, hour
            m_day, m_month, m_year - 1900 }; // 1-based day, 0-based month, year since 1900

    std::time_t time_temp = std::mktime( & time_in );

    // the return value from localtime is a static global - do not call
    // this function from more than one thread!
    //std::tm const *time_out = std::localtime( & time_temp );

    m_day_name = BP_Weekdays(/*time_out->tm_wday*/2);

    //ctor
}

BP_Date_Time::BP_Date_Time(int year, int month, int day, int hour)
{
    m_year = year;
    m_month = month;
    m_day = day;
    m_hour = hour;
    m_day_name = BP_Weekdays(1); // determine default day
}

BP_Date_Time::BP_Date_Time()
{
    //ctor
}

BP_Date_Time::~BP_Date_Time()
{
    //dtor
}

BP_Weekdays BP_Date_Time::get_day_name()
{
    return m_day_name;
}
```

```
int BP_Date_Time::get_day()
{
    return m_day;
}

int BP_Date_Time::get_month()
{
    return m_month;
}

int BP_Date_Time::get_year()
{
    return m_year;
}

int BP_Date_Time::get_hour()
{
    return m_hour;
}

bool BP_Date_Time::operator == (BP_Date_Time rhs)
    {
    if ((m_year == rhs.m_year) &&
        (m_month == rhs.m_month) &&
        (m_day == rhs.m_day) &&
        (m_hour == rhs.m_hour))
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool BP_Date_Time::operator != (BP_Date_Time rhs)
{
    if ((m_year != rhs.m_year) ||
        (m_month != rhs.m_month) ||
        (m_day != rhs.m_day) ||
        (m_hour != rhs.m_hour))
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

```h
#ifndef BP_WEEKDAYS_H
#define BP_WEEKDAYS_H

#include <string>

class BP_Weekdays
{
private:
    std::string m_days[7] =
    {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
    };
    int m_current_day;
public:
    BP_Weekdays (int day); //ctor
    BP_Weekdays (std::string day); //ctor
    BP_Weekdays (const BP_Weekdays &source); // copy ctor
    BP_Weekdays (); //ctor
    ~BP_Weekdays(); //dtor

    BP_Weekdays operator ++ (int);
    BP_Weekdays operator ++ ();

    friend std::ostream& operator << (std::ostream& stream, const BP_Weekdays& day);
};

#endif // BP_WEEKDAYS_H
```

```cpp
#include "BP_Weekdays.h"

#include <iostream>
#include <cstdlib>
#include <boost/algorithm/string.hpp>

std::ostream& operator << (std::ostream& stream, const BP_Weekdays& day)
{
    stream << day.m_days[day.m_current_day];
    return stream;
}

BP_Weekdays::BP_Weekdays (int day) //ctor
{
    if (day > 0 && day <= 7)
    {
        m_current_day = day -1;
    }
    else
    {
        std::cout << "Wrong argument in class constructor"
                  << " of class: \'Weekdays\'. Exiting now..." << std::endl;
        exit(1);
    }
}

BP_Weekdays::BP_Weekdays (std::string day) //ctor
{
    boost::to_upper(day);
    if (day == "SUNDAY") {m_current_day =0;}
    else if (day == "MONDAY") {m_current_day =1;}
    else if (day == "TUESDAY") {m_current_day =2;}
    else if (day == "WEDNESDAY") {m_current_day =3;}
    else if (day == "THURSDAY") {m_current_day =4;}
    else if (day == "FRIDAY") {m_current_day =5;}
    else if (day == "SATURDAY") {m_current_day =6;}
    else
    {
        std::cout << "Wrong argument in class constructor"
                  << " of class: \'Weekdays\'. Exiting now..." << std::endl;
        exit(1);
    }
}

BP_Weekdays::BP_Weekdays (const BP_Weekdays &source)
{
    m_current_day = source.m_current_day;
}

BP_Weekdays::BP_Weekdays () //ctor
{
    m_current_day = 0;
}

BP_Weekdays::~BP_Weekdays()
{
    //dtor
}

BP_Weekdays BP_Weekdays::operator ++ (int)
{
    if (m_current_day == 6)
```

```
        {
            m_current_day = 0;
        }
        else
        {
            m_current_day += 1;
        }
        return *this;
}

BP_Weekdays BP_Weekdays::operator ++ ()
{
    if (m_current_day == 6)
        {
            m_current_day = 0;
        }
        else
        {
            m_current_day += 1;
        }
        return *this;
}
```

```
#ifndef BP_WEATHER_DATA_H_INCLUDED
#define BP_WEATHER_DATA_H_INCLUDED

#include "BP_Date_Time.h"

struct BP_Weather_Data
{
    double m_temp; //temperature (0.1 Celcius)
    BP_Date_Time m_date; // date and time at which the data has been measured

    BP_Weather_Data();
    BP_Weather_Data(std::string line); // reads a line from the weather file and stores the
    relevant data
};


#endif // BP_WEATHER_DATA_H_INCLUDED
```

```cpp
#include <BP_Weather_Data.h>
#include "Trim_And_Cast.h"

#include <fstream>
#include <boost/tokenizer.hpp>
#include <boost/lexical_cast.hpp>

BP_Weather_Data::BP_Weather_Data()
{

}


BP_Weather_Data::BP_Weather_Data(std::string line)
    {
        int yyyymmdd, //year, month and day
            hour; // time of day (1-24)

        boost::char_separator<char> sep(",");
        typedef boost::tokenizer< boost::char_separator<char> > t_tokenizer;
        t_tokenizer tok(line, sep);
        t_tokenizer::iterator token = tok.begin();

        token++; // skip first token STN
        yyyymmdd = trim_and_cast_int(*token);token++;
        hour = trim_and_cast_int(*token);token++;

            // put date and time in the structure
            int year = yyyymmdd/10000;
            int month = (yyyymmdd-year*10000)/100;
            int day = (yyyymmdd-year*10000-month*100);
            m_date = BP_Date_Time(year, month, day, hour);

        token++; // skip DD = Mean wind direction (in degrees) during the 10-minute period
        preceding the time of observation (360=north, 90=east, 180=south, 270=west, 0=calm
        990=variable)
        token++; // skip FH = Hourly mean wind speed (in 0.1 m/s)
        token++; // skip FF = Mean wind speed (in 0.1 m/s) during the 10-minute period
        preceding the time of observation
        token++; // skip FX = Maximum wind gust (in 0.1 m/s) during the hourly division
        m_temp = trim_and_cast_double(*token)/10.0;token++; // temperature (in 1.0 degrees
        Celsius)
        /*token++; // skip T10N = Minimum temperature (in 0.1 degrees Celsius) at 0.1 m in
        the preceding 6-hour period
        token++; // skip TD = Dew point temperature (in 0.1 degrees Celsius) at 1.50 m at
        the time of observation
        token++; // skip SQ = Sunshine duration (in 0.1 hour) during the hourly division,
        calculated from global radiation (-1 for <0.05 hour)
        token++; // skip Q = Global radiation (in J/cm2) during the hourly division
        token++; // skip DR = Precipitation duration (in 0.1 hour) during the hourly division
        token++; // skip RH = Hourly precipitation amount (in 0.1 mm) (-1 for <0.05 mm)
        token++; // skip P = Air pressure (in 0.1 hPa) reduced to mean sea level, at the
        time of observation
        token++; // skip VV = Horizontal visibility at the time of observation (0=less than
        100m, 1=100-200m, 2=200-300m,..., 49=4900-5000m, 50=5-6km, 56=6-7km, 57=7-8km, ...,
        79=29-30km, 80=30-35km, 81=35-40km,..., 89=more than 70km)
        token++; // skip N = Cloud cover (in octants), at the time of observation (9=sky
        invisible)
        token++; // skip U = Relative atmospheric humidity (in percents) at 1.50 m at the
        time of observation
        token++; // skip WW = Present weather code (00-99), description for the hourly
        division. See
```

```
        http://www.knmi.nl/klimatologie/achtergrondinformatie/ww_lijst_engels.pdf
        token++; // skip IX = Indicator present weather code (1=manned and recorded (using
        code from visual observations), 2,3=manned and omitted (no significant weather
        phenomenon to report, not available), 4=automatically recorded (using code from
        visual observations), 5,6=automatically omitted (no significant weather phenomenon
        to report, not available), 7=automatically set (using code from automated
        observations)
        token++; // skip M = Fog 0=no occurrence, 1=occurred during the preceding hour
        and/or at the time of observation
        token++; // skip R = Rainfall 0=no occurrence, 1=occurred during the preceding hour
        and/or at the time of observation
        token++; // skip S = Snow 0=no occurrence, 1=occurred during the preceding hour
        and/or at the time of observation
        token++; // skip O = Thunder  0=no occurrence, 1=occurred during the preceding hour
        and/or at the time of observation
        // skip final token: Y = Ice formation 0=no occurrence, 1=occurred during the
        preceding hour and/or at the time of observation*/
    }
```

```
#ifndef BP_GROUND_PROFILE_H
#define BP_GROUND_PROFILE_H

#include <BP_Indep_State.h>


class BP_Ground_profile : public BP_Indep_State
{
private:
    double m_temperature;
public:
    BP_Ground_profile(double temperature);
    ~BP_Ground_profile();

    bool is_ground_profile();

    double get_temp();
};

#endif // BP_GROUND_PROFILE_H
```

```cpp
#include "BP_Ground_profile.h"

BP_Ground_profile::BP_Ground_profile(double temperature)
{
    m_temperature = temperature;
    //ctor
}

BP_Ground_profile::~BP_Ground_profile()
{
    //dtor
}

bool BP_Ground_profile::is_ground_profile()
{
    return true;
}

double BP_Ground_profile::get_temp()
{
    return m_temperature;
}
```

```
#ifndef BP_WEATHER_PROFILE_H
#define BP_WEATHER_PROFILE_H

#include <string>
#include <vector>

#include "BP_Indep_State.h"
#include "BP_Weather_Data.h"
#include "BP_Date_Time.h"

class BP_Weather_Profile : public BP_Indep_State
{
private:
    std::string m_file_name; // filename of the weather file containing information
    following on the m_next_data structure
    int m_time_step_hour, m_current_time_step; // number of time steps per hour
    unsigned int m_file_position; // current position in weather file 'm_file_name'
    BP_Date_Time m_start_date, m_end_date; // start, current and end date of the simulation
    BP_Weather_Data m_current_data, m_next_data; // data step belonging to current hour and
    next (for interpolation)
    std::vector<BP_Weather_Data*> m_warm_up_data; // vector containing pointers to data of
    weather during warm up period
    int m_warm_up_counter;
    double m_temperature;
public:
    BP_Weather_Profile(BP_Date_Time start, BP_Date_Time finish, int time_step_hour, int
    warm_up_days); // initialises the weather profile to the start date, defines end date
    and how many time steps per hour are used
    ~BP_Weather_Profile();

    void next_time_step();
    void next_warm_up_step();
    BP_Weekdays get_day_name();
    int get_year();
    int get_month();
    int get_day();
    int get_hour();
    double get_temp();
    void update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd& B,
    Eigen::VectorXd& u);

    bool is_weather_profile();
    bool finished();

    void tester();
};

#endif // BP_WEATHER_PROFILE_H
```

```cpp
#include "BP_Weather_Profile.h"
#include "Trim_And_Cast.h"
#include <iostream>
#include <cstdlib>

#include <fstream>
#include <boost/tokenizer.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/algorithm/string/trim.hpp>


std::string weather_file(const int& year)
{
    int m_year = year-((year-1)%10);

    switch (m_year)
    {
    case 1981:
        return "uurgeg_260_1981-1990.txt";
        break;
    case 1991:
        return "uurgeg_260_1991-2000.txt";
        break;
    case 2001:
        return "uurgeg_260_2001-2010.txt";
        break;
    case 2011:
        return "uurgeg_260_2011-2020.txt";
        break;
    default:
        std::cout << "Error in reading date for weather files! Exiting now..." << std::endl;
        exit(1);
        return "error";
    }
}

std::string read_next_line(std::string& file_name, const int& current_year, unsigned int&
position) // caution! this function updates some arguments by reference // this still needs
some functionality for the case when an empty line gets passed to it
{
    std::fstream input(file_name.c_str()); // open a file stream
    std::string line; // initialise a string to hold lines being read from file

    if (position == 0)
    {
        for (int i = 0; i < 34; i++) // skip the first 33 lines (Heading of the file), the
        34th contains the first line of data
        {
            getline(input, line);
        } // the 34th line is now stored in 'std::string line'
        position = input.tellg(); // update the position in the file
    }
    else
    {
        input.seekg(position); // this is correct, assuming the current position is at the
        start of a line and not in the heading
        getline(input,line); // get the line at this position
        if(input.eof()) // if this line is the last in the file, then update the file name
        and position
        {
            file_name = weather_file(current_year+1);
            position = 0;
```

```
        }
        else // otherwise only update the position
        {
            position = input.tellg();
        }
    }
    return line;
}

BP_Weather_Profile::BP_Weather_Profile(BP_Date_Time start, BP_Date_Time finish, int
time_step_hour, int warm_up_days) : BP_Indep_State()
{
    m_start_date = start;
    m_end_date = finish;
    m_time_step_hour = time_step_hour;
    m_current_time_step = 0;
    m_file_position = 0;
    m_warm_up_counter = warm_up_days*24;

    m_file_name = weather_file(m_start_date.get_year()); // find the file that contains the
    given year
    m_current_data = BP_Weather_Data(read_next_line(m_file_name, start.get_year(),
    m_file_position)); // read first line of the first file

    int yyyymmdd, //year, month and day
            hour; // time of day (1-24)

    std::fstream input(m_file_name.c_str());
    std::string line;
    input.seekg(m_file_position);
    boost::char_separator<char> sep(",");
    typedef boost::tokenizer< boost::char_separator<char> > t_tokenizer;

    while (m_start_date != m_current_data.m_date) // keep reading lines in the file until
    the correct line has been found, this is a time consuming approach (15 seconds if the
    starting date is the last file entry)
    {
        m_file_position = input.tellg(); // position before getline, because when it is the
        searched line the position must be stored
        getline(input, line); // get next line from the file
        t_tokenizer tok(line, sep);
        t_tokenizer::iterator token = tok.begin();

        token++; // skip first token STN
        yyyymmdd = trim_and_cast_int(*token);token++; // date
        hour = trim_and_cast_int(*token); // time

        // put date and time in the structure
        int year = yyyymmdd/10000;
        int month = (yyyymmdd-year*10000)/100;
        int day = (yyyymmdd-year*10000-month*100);
        m_current_data.m_date = BP_Date_Time(year, month, day, hour);
    }

    m_current_data = BP_Weather_Data(read_next_line(m_file_name, start.get_year(),
    m_file_position)); // parse the line containing the data of the start time
    m_next_data = BP_Weather_Data(read_next_line(m_file_name,
    m_current_data.m_date.get_year(), m_file_position)); // parse next line, for
    interpolation purposes

    if(warm_up_days > 0)
    {
```

```
        m_warm_up_data.push_back(&m_current_data);
        m_warm_up_data.push_back(&m_next_data);
    }

    unsigned int file_pos_warm_up = m_file_position;
    std::string file_name_warm_up = m_file_name;

    for (int i = 2; i < warm_up_days*24+1; i++)
    {
        m_warm_up_data.push_back(new BP_Weather_Data(read_next_line(file_name_warm_up,
        m_warm_up_data[i-1]->m_date.get_year(), file_pos_warm_up)));
    }

    if (warm_up_days == 0)
    {
        m_temperature = m_current_data.m_temp + (m_next_data.m_temp-m_current_data.m_temp) *
        (m_current_time_step/boost::lexical_cast<double>(m_time_step_hour)); //
        interpolation regarding time steps
    }
    else if (warm_up_days > 0)
    {
        m_temperature = m_warm_up_data[m_warm_up_counter]->m_temp +
        (m_warm_up_data[m_warm_up_counter-1]->m_temp -
        m_warm_up_data[m_warm_up_counter]->m_temp) *
        (m_current_time_step/boost::lexical_cast<double>(m_time_step_hour)); //
        interpolation regarding time steps
    }
    else
    {
        std::cout << "Error in reading warm up days, exiting.." << std::endl;
        exit(1);
    }
    //ctor
}


BP_Weather_Profile::~BP_Weather_Profile()
{
    for (unsigned int i = 2; i < m_warm_up_data.size(); i++) // delete pointers in the
    vector (first 2 are not allocated ptrs but current_ and next_data)
    {
        delete m_warm_up_data[i];
    }
    m_warm_up_data.clear(); // clear the vector that now has some pointers to released memory
    //dtor
}

BP_Weekdays BP_Weather_Profile::get_day_name()
{
    return m_current_data.m_date.get_day_name();
}

int BP_Weather_Profile::get_year()
{
    return m_current_data.m_date.get_year();
}

int BP_Weather_Profile::get_month()
{
    return m_current_data.m_date.get_month();
}
```

```
int BP_Weather_Profile::get_day()
{
    return m_current_data.m_date.get_day();
}

int BP_Weather_Profile::get_hour()
{
    return m_current_data.m_date.get_hour();
}

double BP_Weather_Profile::get_temp()
{
    return m_temperature;
}

void BP_Weather_Profile::next_time_step()
{
    m_current_time_step++;
    if(m_current_time_step == m_time_step_hour)
    {
        m_current_time_step = 0;
        m_current_data = m_next_data;
        m_next_data = BP_Weather_Data(read_next_line(m_file_name,
        m_current_data.m_date.get_year(), m_file_position)); // read in the next data line,
        for interpolation purposes
    }

    m_temperature = m_current_data.m_temp + (m_next_data.m_temp-m_current_data.m_temp) *
    (m_current_time_step/boost::lexical_cast<double>(m_time_step_hour)); // interpolation
    regarding time steps
}

void BP_Weather_Profile::next_warm_up_step()
{
    m_current_time_step++;
    if(m_current_time_step == m_time_step_hour)
    {
        m_current_time_step = 0;
        m_warm_up_counter--;
    }
    if (m_warm_up_counter != 0)
    {
        m_temperature = m_warm_up_data[m_warm_up_counter]->m_temp +
        (m_warm_up_data[m_warm_up_counter-1]->m_temp -
        m_warm_up_data[m_warm_up_counter]->m_temp) *
        (m_current_time_step/boost::lexical_cast<double>(m_time_step_hour)); //
        interpolation regarding time steps
    }
    else if (m_warm_up_counter == 0)
    {
        m_temperature = m_warm_up_data[m_warm_up_counter]->m_temp;
    }
}

bool BP_Weather_Profile::is_weather_profile()
{
    return true;
}

void BP_Weather_Profile::update_sys(Eigen::MatrixXd& A, Eigen::VectorXd& x, Eigen::MatrixXd&
B, Eigen::VectorXd& u)
{
```

```
    u(m_index) = m_temperature;
}

void BP_Weather_Profile::tester()
{
    //std::cout << "Time_step: " << m_current_time_step << std::endl;
    //std::cout << "Date and time: " << m_current_data.m_date.get_year() << "/" <<
    m_current_data.m_date.get_month() << "/" << m_current_data.m_date.get_day() << " - " <<
    m_current_data.m_date.get_hour() << std::endl;
    //std::cout << "Temperature: " << BP_State::get_temp() << std::endl << std::endl;
}

bool BP_Weather_Profile::finished()
{
    if (m_current_data.m_date == m_end_date)
    {
        std::cout << "Current - Year: " << m_current_data.m_date.get_year() << ", Month: "
        << m_current_data.m_date.get_month() << ", Day: " << m_current_data.m_date.get_day()
        << ", Hour: " << m_current_data.m_date.get_hour() << std::endl;
        std::cout << "End - Year: " << m_end_date.get_year() << ", Month: " <<
        m_end_date.get_month() << ", Day: " << m_end_date.get_day() << ", Hour: " <<
        m_end_date.get_hour() << std::endl;

        return true;
    }
    else
    {
        return false;
    }
}
```

```
#ifndef BP_SIMULATION_H
#define BP_SIMULATION_H

#include "BP_Space.h"
#include "BP_Wall.h"
#include "BP_window.h"
#include "BP_Floor.h"
#include "BP_Construction.h"
#include "BP_Weather_Profile.h"
#include "BP_Ground_profile.h"

#include <Eigen/Core>
#include <vector>


// this class still needs a copy constructor, assignment operator and a destructor (release
allocated memory)??
class BP_Simulation
{
private:
    std::vector<BP_Dep_State*> m_dep_states; // collection of ptrs to dependant states in
    the system
    std::vector<BP_Indep_State*> m_indep_states; // collection of ptrs to independant states
    in the system

    std::vector<BP_Material> m_materials; // vector of objects of the BP_Material structure
    (structure is defined in BP_Construction.h)
    std::vector<BP_Construction> m_constructions; // vector of objects of the
    BP_Construction structure
    std::vector<BP_Space*> m_space_ptrs; // vector of pointers to objects of the BP_Space
    class (pointers since adresses of objects should not change when vector size changes)
    std::vector<BP_Wall*> m_wall_ptrs; // vector of pointers to objects of the BP_Wall class
    (pointers since adresses of objects should not change when vector size changes)
    std::vector<BP_Window*> m_window_ptrs; // vector of pointers to objects of the BP_Window
    class (pointers since adresses of objects should not change when vector size changes)
    std::vector<BP_Floor*> m_floor_ptrs; // vector of pointers to objects of the BP_Wall
    class (pointers since adresses of objects should not change when vector size changes)
    BP_Weather_Profile* m_weather_profile;
    BP_Ground_profile* m_ground_profile;

    int m_time_step_hour, m_warm_up_days;
    int m_current_time_step;

    Eigen::VectorXd m_SS_x, m_SS_u; // vectors to hold the states, x dependant states, u
    independant states
    Eigen::MatrixXd m_SS_init_A, m_SS_init_B; // the state matrices, A (dependant) and B
    (independant) input matrices, C (dependant) and D (independant) output matrices
    Eigen::MatrixXd m_SS_A, m_SS_B; // the state matrices, A (dependant) and B (independant)
    input matrices, C (dependant) and D (independant) output matrices
public:
    BP_Simulation(std::string file_name);
    ~BP_Simulation();

    void test_values(); // for testing purposes

    void ODE_function(const Eigen::VectorXd &x, Eigen::VectorXd &dxdt, double);
    //vo
    void next_warm_up_step();
    void sim_warm_up_period();
    void next_time_step(); // moves to next time step and solves the ODE system to update
    the state temperatures
    void sim_period();
```

```
};

#endif // BP_SIMULATION_H
```

```cpp
#include "BP_Simulation.h"
#include "Trim_And_Cast.h"
#include "BP_Date_Time.h"
//#include "BP_State.h"

#include <fstream>
#include <boost/bind.hpp>
#include <boost/tokenizer.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/algorithm/string/trim.hpp>
#include <boost/numeric/odeint.hpp>
#include <boost/numeric/odeint/external/eigen/eigen_algebra.hpp>


namespace odeint = boost::numeric::odeint; // shorten name space declaration a bit
typedef odeint::runge_kutta4<Eigen::VectorXd, double, Eigen::VectorXd, double,
odeint::vector_space_algebra> stepper; // this defines what solver is used and what type of
variable is used for the states

BP_Simulation::BP_Simulation(std::string file_name)
{
    m_current_time_step = 0;

    std::fstream input(file_name.c_str()); // open a file stream
    std::string line; // initialise a string to hold lines being read from file
    boost::char_separator<char> sep(","); // defines what separates tokens in a string
    typedef boost::tokenizer< boost::char_separator<char> > t_tokenizer; // settings for the
    boost::tokenizer
    char type_ID; // holds information about what type of information is described by the
    line currently read

    while(!input.eof())
    {
        getline(input, line); // get the next line from the file
        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)
        if (line == "") //skip empty lines (tokenizer does not like it)
        {
            continue; // continue to next line
        }
        t_tokenizer tok(line, sep); // tokenize the line
        t_tokenizer::iterator token = tok.begin(); // set iterator to first token
        type_ID = trim_and_cast_char(*token); // interpret first token as type ID

        switch (type_ID)
        {
        case 'U':
            {
                token++; // next token holds number of warm up days
                m_warm_up_days = trim_and_cast_int(*token);
                break;
            }
        case 'T':
            {
                token++; //next token holds number of time steps per hour
                m_time_step_hour = trim_and_cast_int(*token);
                break;
            }
        case 'E':
            {
                int year, month, day, hour; // initialize variables to temporarily hold data
                std::string day_name; // initialize string to temporarily hold day name
```

```
            token++; // start year
            year = trim_and_cast_int(*token);
            token++; // start month
            month = trim_and_cast_int(*token);
            token++; // start day
            day = trim_and_cast_int(*token);
            token++; // start day name
            day_name = *token;
            boost::algorithm::trim(day_name); // remove white space in front and at end
            of string
            token++; // start hour
            hour = trim_and_cast_int(*token);

            BP_Date_Time start(year, month, day, hour, "default");

            token++; // finish year
            year = trim_and_cast_int(*token);
            token++; // finish month
            month = trim_and_cast_int(*token);
            token++; // finish day
            day = trim_and_cast_int(*token);
            token++; // finish hour
            hour = trim_and_cast_int(*token);
            std::cout << year << ", " << month << ", " << day << ", " << hour <<
            std::endl;
            BP_Date_Time finish(year, month, day, hour);

            m_weather_profile = new BP_Weather_Profile(start, finish, m_time_step_hour,
            m_warm_up_days); // it is important that the nr of warm up days is declared
            before the weather profile (consider change in future)
            m_indep_states.push_back(m_weather_profile);
            break;
        }
    case 'G':
        {
            double temperature;
            token++; // Constant ground temperature
            temperature = trim_and_cast_double(*token);
            m_ground_profile = new BP_Ground_profile(temperature);
            m_indep_states.push_back(m_ground_profile);
            break;
        }
    case 'M':
        {
            BP_Material material; // initialize to temporarily hold data
            token++; // Material_ID
            material.m_material_ID = *token;
            boost::algorithm::trim(material.m_material_ID);
            token++; // material name
            material.m_name = *token;
            boost::algorithm::trim(material.m_name);
            token++; // specific weight
            material.m_spec_weight = trim_and_cast_double(*token);
            token++; // specific heat
            material.m_spec_heat = trim_and_cast_double(*token);
            token++; // thermal conductivity
            material.m_therm_conductivity = trim_and_cast_double(*token);
            m_materials.push_back(material);
            break;
        }
    case 'C':
```

```
{
    BP_Construction construction; // initialize to temporarily hold data
    token++; // Construction_ID
    construction.m_construction_ID = *token;
    boost::algorithm::trim(construction.m_construction_ID); // remove white
    space at front and end of string
    std::vector<std::string> layer_mat_ID; // will hold material ID of all layers
    std::vector<double> layer_thicknesses; // will hold thicknesses of all layers
    token++; // material ID
    while (token != tok.end()) // read indefinite number of layers into the
    construction
    {
        std::string mat_ID = *token;
        boost::algorithm::trim(mat_ID);
        layer_mat_ID.push_back(mat_ID);
        token++; // layer thickness
        layer_thicknesses.push_back(trim_and_cast_double(*token));
        token++; // Material ID or final token
    }

    if (layer_mat_ID.size() != layer_thicknesses.size())
    {
        std::cout << "Error in reading construction info, exiting..." <<
        std::endl;
        exit(1);
    }
    BP_Con_Layer layer_entry; // initialize a layer structure to temporarily
    hold data

    construction.m_total_thickness = 0; // initialise to zero
    for (unsigned int i = 0; i < layer_mat_ID.size(); i++)
    {
        for (unsigned int j = 0; j < m_materials.size(); j++)
        {
            if (layer_mat_ID[i] == m_materials[j].m_material_ID)
            {
                layer_entry.m_material = m_materials[j];
                layer_entry.m_thickness = layer_thicknesses[i];
                construction.m_layers.push_back(layer_entry);
                construction.m_total_thickness += layer_thicknesses[i];
                break;
            }
            else if (j == (m_materials.size() - 1))
            {
                std::cout << "Error in processing construction info, exiting..."
                << std::endl;
            }
        }
    }

    //compute capacitance
    construction.m_capacitance_per_area = 0; // initialise value to zero
    for (unsigned int i = 0; i < construction.m_layers.size(); i++) // add each
    layer's contribution
    {
        construction.m_capacitance_per_area +=
        (construction.m_layers[i].m_thickness/1000) *
        construction.m_layers[i].m_material.m_spec_heat *
        construction.m_layers[i].m_material.m_spec_weight; // capacitance equals
        thickness*specific heat* speciic weight
    }
```

```
//location of measure point
construction.m_measure_point = 0.5; // relative location, initialise to
centre of wall, may need to be with respect to equal capacitance at each side?
double abs_measure_point = construction.m_measure_point *
construction.m_total_thickness; // the absolute value of the distance from
surface on side one to the temperature measure point

construction.m_resistance_to_side_1 = 0; //initialise value to zero
construction.m_resistance_to_side_2 = 0; //initialise value to zero

// compute resistances (per square metre) to each side
if (construction.m_measure_point >= 0.0 && construction.m_measure_point <=
1.0) // see if the measure point lies within the construction
{
    for (unsigned int i = 0; i < construction.m_layers.size(); i++) // cycle
    through each layer
    {
        // resistance to side 1
        if (construction.m_layers[i].m_thickness <= abs_measure_point) // if
        true, this layer contributes to the resistance to flux to side 1
        {
            construction.m_resistance_to_side_1 +=
            (construction.m_layers[i].m_thickness/1000)/(construction.m_layers
            [i].m_material.m_therm_conductivity); // resistance per square
            metre equals thickness(m)/thermal conductivity
            abs_measure_point -= construction.m_layers[i].m_thickness; //
            decrement the distance to the temperature measure point
        }
        else if (abs_measure_point > 0) // if true, part of this layer
        contributes to resistance to flux to side 1 and the other to the
        resistance of flux to side 2
        {
            construction.m_resistance_to_side_1 +=
            (abs_measure_point/1000)/(construction.m_layers[i].m_material.m_th
            erm_conductivity); // add resistance at side 1 of the measure
            point
            abs_measure_point -= construction.m_layers[i].m_thickness; //
            decrement distance to measure point by layer thickness to add
            remainder of the current layer to the resistance to flux to side
            two
            construction.m_resistance_to_side_2 +=
            ((-1.0*abs_measure_point)/1000)/(construction.m_layers[i].m_materi
            al.m_therm_conductivity); // add resistance at side 2 of the
            measure point
        }
        else // otherwise this layer (and the remainder) contributes to the
        resistance to flux to side 2
        {
            construction.m_resistance_to_side_2 +=
            (construction.m_layers[i].m_thickness/1000)/(construction.m_layers
            [i].m_material.m_therm_conductivity); // the remaining layers
            contribute to the resistance to flux to side two of the measure
            point
        }
    }
}
else
{
    std::cout << "Temperature measure point is outside construction,
    exiting...." << std::endl;
    exit(1);
}
```

```
            m_constructions.push_back(construction);
            break;
        }
    case 'S':
        {
            double volume, heating_capacity, cooling_capacity, heat_set_point,
            cool_set_point, ACH;
            token++; //space_ID
            std::string space_ID = *token;
            boost::algorithm::trim(space_ID);
            token++; //volume
            volume = trim_and_cast_double(*token);
            token++; // heating capacity
            heating_capacity = trim_and_cast_double(*token);
            token++; // cooling capacity
            cooling_capacity = trim_and_cast_double(*token);
            token++; // heating set point
            heat_set_point = trim_and_cast_double(*token);
            token++; // cooling set point
            cool_set_point = trim_and_cast_double(*token);
            token++; // air changes per hour
            ACH = trim_and_cast_double(*token);

            BP_Space* space_ptr = new BP_Space(space_ID, volume, heating_capacity,
            cooling_capacity, heat_set_point, cool_set_point, ACH, m_weather_profile);
            // initialise a ptr to an object of the space class with the values read
            from the file
            m_space_ptrs.push_back(space_ptr); // add the space pointer to the space
            vector
            m_dep_states.push_back(space_ptr);
            break;
        }
    case 'W':
        {
            token++; // Wall_ID
            std::string wall_ID = *token;
            boost::algorithm::trim(wall_ID);
            token++; // Construction_ID
            std::string construction_ID = *token;
            boost::algorithm::trim(construction_ID);
            BP_Construction* construction_ptr = NULL;
            for (unsigned int i = 0; i < m_constructions.size(); i++)
            {
                if (construction_ID == m_constructions[i].m_construction_ID)
                {
                    construction_ptr = &m_constructions[i];
                }
            }
            token++; // Surface area
            double area = trim_and_cast_double(*token);
            token++; // Surface Orientation, skipped for now
            token++; // Space ID side one
            std::string space_ID_1 = *token;
            boost::algorithm::trim(space_ID_1);
            token++; // SPace Id side two
            std::string space_ID_2 = *token;
            boost::algorithm::trim(space_ID_2);

            BP_State* state_ptr_side_1;
            BP_State* state_ptr_side_2;
            bool side_1_found = false;
```

```
            bool side_2_found = false;

            for (unsigned int i = 0; i < m_space_ptrs.size(); i ++)
            {
                if(space_ID_1 == m_space_ptrs[i]->get_ID())
                {
                    state_ptr_side_1 = m_space_ptrs[i];
                    side_1_found = true;
                }
                else if (space_ID_1 == "E")
                {
                    state_ptr_side_1 = m_weather_profile;
                    side_1_found = true;
                }
                else if (space_ID_1 == "G")
                {
                    state_ptr_side_1 = m_ground_profile;
                    side_1_found = true;
                }
                if(space_ID_2 == m_space_ptrs[i]->get_ID())
                {
                    state_ptr_side_2 = m_space_ptrs[i];
                    side_2_found = true;
                }
                else if (space_ID_2 == "E")
                {
                    state_ptr_side_2 = m_weather_profile;
                    side_2_found = true;
                }
                else if (space_ID_2 == "G")
                {
                    state_ptr_side_2 = m_ground_profile;
                    side_2_found = true;
                }
            }

            if (!side_1_found || !side_2_found)
            {
                std::cout << "Error in assigning neighbouring states to wall: " <<
                wall_ID << ". Exiting now..." << std::endl;
                exit(1);
            }

            BP_Wall* wall_ptr = new BP_Wall(wall_ID, area, construction_ptr,
            state_ptr_side_1, state_ptr_side_2);
            m_wall_ptrs.push_back(wall_ptr); // add the wall pointer to the wall vector
            m_dep_states.push_back(wall_ptr);
            break;
        }
    case 'V':
        {
            token++; // window_ID
            std::string window_ID = *token;
            boost::algorithm::trim(window_ID);
            token++; // area
            double area = trim_and_cast_double(*token);
            token++; // U_value
            double U_value = trim_and_cast_double(*token);
            token++; // capacitance_per_area
            double capacitance_per_area = trim_and_cast_double(*token);
            token++; // skipped for now
            token++; // Space ID side one
```

```
                std::string space_ID_1 = *token;
                boost::algorithm::trim(space_ID_1);
                token++; // SPace Id side two
                std::string space_ID_2 = *token;
                boost::algorithm::trim(space_ID_2);

                BP_State* state_ptr_side_1;
                BP_State* state_ptr_side_2;
                bool side_1_found = false;
                bool side_2_found = false;

                for (unsigned int i = 0; i < m_space_ptrs.size(); i ++)
                {
                    if(space_ID_1 == m_space_ptrs[i]->get_ID())
                    {
                        state_ptr_side_1 = m_space_ptrs[i];
                        side_1_found = true;
                    }
                    else if (space_ID_1 == "E")
                    {
                        state_ptr_side_1 = m_weather_profile;
                        side_1_found = true;
                    }
                    else if (space_ID_1 == "G")
                    {
                        state_ptr_side_1 = m_ground_profile;
                        side_1_found = true;
                    }

                    if(space_ID_2 == m_space_ptrs[i]->get_ID())
                    {
                        state_ptr_side_2 = m_space_ptrs[i];
                        side_2_found = true;
                    }
                    else if (space_ID_2 == "E")
                    {
                        state_ptr_side_2 = m_weather_profile;
                        side_2_found = true;
                    }
                    else if (space_ID_2 == "G")
                    {
                        state_ptr_side_2 = m_ground_profile;
                        side_2_found = true;
                    }
                }

                if (!side_1_found || !side_2_found)
                {
                    std::cout << "Error in assigning neighbouring states to wall, exiting
                    now..." << std::endl;
                    exit(1);
                }

                BP_Window* window_ptr = new BP_Window(window_ID, area, U_value,
                capacitance_per_area, state_ptr_side_1, state_ptr_side_2);
                m_window_ptrs.push_back(window_ptr); // add the wall pointer to the wall
                vector
                m_dep_states.push_back(window_ptr);
                break;
            }
        case 'F':
            {
```

```
                token++; // flooor_ID
                std::string floor_ID = *token;
                boost::algorithm::trim(floor_ID);
                token++; // Construction_ID
                std::string construction_ID = *token;
                boost::algorithm::trim(construction_ID);
                BP_Construction* construction_ptr = NULL;
                for (unsigned int i = 0; i < m_constructions.size(); i++)
                {
                    if (construction_ID == m_constructions[i].m_construction_ID)
                    {
                        construction_ptr = &m_constructions[i];
                    }
                }
                token++; // Surface area
                double area = trim_and_cast_double(*token);
                token++; // Space ID side one
                std::string space_ID_1 = *token;
                boost::algorithm::trim(space_ID_1);
                token++; // SPace Id side two
                std::string space_ID_2 = *token;
                boost::algorithm::trim(space_ID_2);

                BP_State* state_ptr_side_1;
                BP_State* state_ptr_side_2;
                bool side_1_found = false;
                bool side_2_found = false;

                for (unsigned int i = 0; i < m_space_ptrs.size(); i ++)
                {
                    if(space_ID_1 == m_space_ptrs[i]->get_ID())
                    {
                        state_ptr_side_1 = m_space_ptrs[i];
                        side_1_found = true;
                    }
                    else if (space_ID_1 == "E")
                    {
                        state_ptr_side_1 = m_weather_profile;
                        side_1_found = true;
                    }
                    else if (space_ID_1 == "G")
                    {
                        state_ptr_side_1 = m_ground_profile;
                        side_1_found = true;
                    }
                    if(space_ID_2 == m_space_ptrs[i]->get_ID())
                    {
                        state_ptr_side_2 = m_space_ptrs[i];
                        side_2_found = true;
                    }
                    else if (space_ID_2 == "E")
                    {
                        state_ptr_side_2 = m_weather_profile;
                        side_2_found = true;
                    }
                    else if (space_ID_2 == "G")
                    {
                        state_ptr_side_2 = m_ground_profile;
                        side_2_found = true;
                    }
                }
```

```
                if (!side_1_found || !side_2_found)
                {
                    std::cout << "Error in assigning neighbouring states to floor, exiting
                    now..." << std::endl;
                    exit(1);
                }

                BP_Floor* floor_ptr = new BP_Floor(floor_ID, area, construction_ptr,
                state_ptr_side_1, state_ptr_side_2);
                m_floor_ptrs.push_back(floor_ptr); // add the floor pointer to the floor
                vector
                m_dep_states.push_back(floor_ptr);
            }
            break;
        default:
            break;
        }
    }

    // initialize the state vectors and matrices to their correct sizes
    m_SS_u = Eigen::VectorXd::Constant(m_weather_profile->get_count(), 1.0); // seed state
    vector u with initial values (must be initiated to 1!)
    m_SS_x = Eigen::VectorXd::Constant(m_space_ptrs[0]->get_count(), 0.0); // seed state
    vector x with initial values
    m_SS_init_A = Eigen::MatrixXd::Zero(m_space_ptrs[0]->get_count(),
    m_space_ptrs[0]->get_count()); // seed state matrix A with initial values
    m_SS_init_B = Eigen::MatrixXd::Zero(m_space_ptrs[0]->get_count(),
    m_weather_profile->get_count()); // seed state matrix B with initial values

    for (unsigned int i = 0; i < m_indep_states.size(); i++)
    {
        m_SS_u(m_indep_states[i]->get_index()) = m_indep_states[i]->get_temp();
    }

    for (unsigned int i = 0; i < m_dep_states.size(); i++)
    {
        m_dep_states[i]->init_sys(m_SS_init_A, m_SS_init_B);
    }

    // create/erase the output file and set the heading of the comma separated file
    std::ofstream output; // create a stream
    output.open("Simulation_output.txt"); // create/erase the output file

    output << "Day Name, Year, Month, Day, Time, Te, "; // set the heading
    for (unsigned int i = 0; i < m_space_ptrs.size(); i++) // set heading for the spaces
    {
        output << "T_space_" + m_space_ptrs[i]->get_ID() << ", ";
    }
    for (unsigned int i = 0; i < m_wall_ptrs.size(); i++) // set the heading for the walls
    {
        output << "T_wall_" + m_wall_ptrs[i]->get_ID() << ", ";
    }
    for (unsigned int i = 0; i < m_floor_ptrs.size(); i++) // set the heading for the floors
    {
        output << "T_floor_" + m_floor_ptrs[i]->get_ID();
        if (i != m_floor_ptrs.size()-1) {output << ", ";}
    }
    output << std::endl; // end the heading line
    output.close(); // close the output file
    //ctor
}
```

```
BP_Simulation::~BP_Simulation()
{
    delete m_weather_profile;
    delete m_ground_profile;

    for (unsigned int i = 0; i < m_space_ptrs.size(); i++)
    {
        delete m_space_ptrs[i];
        m_space_ptrs.clear();
    }

    for (unsigned int i = 0; i < m_wall_ptrs.size(); i++)
    {
        delete m_wall_ptrs[i];
        m_wall_ptrs.clear();
    }

    for (unsigned int i = 0; i < m_window_ptrs.size(); i++)
    {
        delete m_window_ptrs[i];
        m_window_ptrs.clear();
    }

    for (unsigned int i = 0; i < m_floor_ptrs.size(); i++)
    {
        delete m_floor_ptrs[i];
        m_floor_ptrs.clear();
    }
    //dtor
}

void BP_Simulation::ODE_function(const Eigen::VectorXd &x, Eigen::VectorXd &dxdt, double) //
x' = A*x + B*u
{
    dxdt = m_SS_A*x + m_SS_B*m_SS_u; // x is passed as a function argument as it must be
    varied by the ODE solver
}

void BP_Simulation::test_values() // for testing purposes
{
    /*std::cout << m_SS_A << std::endl <<std::endl;
    std::cout << m_SS_x << std::endl <<std::endl;
    std::cout << m_SS_B << std::endl <<std::endl;
    std::cout << m_SS_u << std::endl <<std::endl;*/

    m_wall_ptrs[0]->get_res();
    std::cout << m_wall_ptrs[0]->get_capa() << std::endl << std::endl;

    m_space_ptrs[0]->get_res();
    std::cout << m_space_ptrs[0]->get_capa() << std::endl;
}

void BP_Simulation::next_warm_up_step()
{
    m_current_time_step++; // increment time step counter
    m_weather_profile->next_warm_up_step(); // increment time step in weather profile

    if(m_current_time_step == m_time_step_hour) // if final time step of the hour has been
    reached, reset the time step counter
    {
        m_current_time_step = 0; // reset time step counter
    }
```

```
    m_SS_A = m_SS_init_A;
    m_SS_B = m_SS_init_B;

    for (unsigned int i = 0; i < m_indep_states.size(); i++)
    {
        m_indep_states[i]->update_sys(m_SS_A, m_SS_x, m_SS_B, m_SS_u);
    }

    for (unsigned int i = 0; i < m_dep_states.size(); i++)
    {
        m_dep_states[i]->update_sys(m_SS_A, m_SS_x, m_SS_B, m_SS_u,
        (3600.0/(m_time_step_hour)));
    }

    // Call Boost::ODEint function to solve the state space system over the incremented time
    step
    // arguments:   definition of the stepper (predefined at top of source file),
    //              ODE_function, because it is a member function it should be bound to a
    temporary function which is then passed as argument,
    //              the state vector x, begin time, end time, time increments used by solver
    (not to be confused with time steps of building physics simulation)
    //
    odeint::integrate_adaptive(stepper(), boost::bind(&BP_Simulation::ODE_function, this,
    _1, _2, _3), m_SS_x, 0.0, (3600.0/(m_time_step_hour)), (3600.0/(m_time_step_hour*1)));
    // solves the state space system over the duration of one time step
}

void BP_Simulation::sim_warm_up_period()
{
    for (int i = 0; i < (m_warm_up_days*24*m_time_step_hour); i++)
    {
        this->next_warm_up_step();
    }

    if (m_current_time_step == 0)
    {
        std::ofstream output;
        output.open("Simulation_output.txt", std::ios::app);

        output << m_weather_profile->get_day_name() << ", "
               << m_weather_profile->get_year() << ", "
               << m_weather_profile->get_month() << ", "
               << m_weather_profile->get_day() << ", "
               << m_weather_profile->get_hour() << ", "
               << m_SS_u(m_weather_profile->get_index()) << ", ";
        for (unsigned int i = 0; i < m_space_ptrs.size(); i++)
        {
            output << m_SS_x(m_space_ptrs[i]->get_index()) << ", ";
        }
        for (unsigned int i = 0; i < m_wall_ptrs.size(); i++)
        {
            output << m_SS_x(m_wall_ptrs[i]->get_index()) << ", ";
        }
        for (unsigned int i = 0; i < m_floor_ptrs.size(); i++)
        {
            output << m_SS_x(m_floor_ptrs[i]->get_index());
            if (i != m_floor_ptrs.size()-1) {output << ", ";}
        }
        output << std::endl;
        output.close();
    }
```

```
}

void BP_Simulation::next_time_step() // increment time step in simulation
{
    m_current_time_step++; // increment time step counter
    m_weather_profile->next_time_step(); // increment time step in weather profile

    if(m_current_time_step == m_time_step_hour) // if final time step of the hour has been
    reached, reset the time step counter
    {
        m_current_time_step = 0; // reset time step counter
    }

    m_SS_A = m_SS_init_A;
    m_SS_B = m_SS_init_B;

    for (unsigned int i = 0; i < m_indep_states.size(); i++)
    {
        m_indep_states[i]->update_sys(m_SS_A, m_SS_x, m_SS_B, m_SS_u);
    }

    for (unsigned int i = 0; i < m_dep_states.size(); i++)
    {
        m_dep_states[i]->update_sys(m_SS_A, m_SS_x, m_SS_B, m_SS_u,
        (3600.0/(m_time_step_hour)));
    }

    // Call Boost::ODEint function to solve the state space system over the incremented time
    step
    // arguments:   definition of the stepper (predefined at top of source file),
    //              ODE_function, because it is a member function it should be bound to a
    temporary function which is then passed as argument,
    //              the state vector x, begin time, end time, time increments used by solver
    (not to be confused with time steps of building physics simulation)
    //
    odeint::integrate_adaptive(stepper(), boost::bind(&BP_Simulation::ODE_function, this,
    _1, _2, _3), m_SS_x, 0.0, (3600.0/(m_time_step_hour)), (3600.0/(m_time_step_hour*1)));
    // solves the state space system over the duration of one time step

    std::ofstream output;
    output.open("Simulation_output.txt", std::ios::app);

    if (m_current_time_step == 0)
    {
        output << m_weather_profile->get_day_name()
               << ", " << m_weather_profile->get_year()
               << ", " << m_weather_profile->get_month()
               << ", " << m_weather_profile->get_day()
               << ", " << m_weather_profile->get_hour()
               << ", " << m_SS_u(m_weather_profile->get_index()) << ", ";
        for (unsigned int i = 0; i < m_space_ptrs.size(); i++)
        {
            output << m_SS_x(m_space_ptrs[i]->get_index()) << ", ";
        }
        for (unsigned int i = 0; i < m_wall_ptrs.size(); i++)
        {
            output << m_SS_x(m_wall_ptrs[i]->get_index()) << ", ";
        }
        for (unsigned int i = 0; i < m_floor_ptrs.size(); i++)
        {
            output << m_SS_x(m_floor_ptrs[i]->get_index());
```

```
            if (i != m_floor_ptrs.size()-1) {output << ", ";}
        }
        output << std::endl;
    }
    output.close();

}

void BP_Simulation::sim_period()
{
    sim_warm_up_period();

    for (unsigned int i = 0; i < m_space_ptrs.size(); i++)
    {
        m_space_ptrs[i]->set_power_count_zero();
    }

    while (!m_weather_profile->finished())
    {
        for (int i = 0; i < m_time_step_hour; i++)
        {
            next_time_step();
        }
    }

    for (unsigned int i = 0; i < m_space_ptrs.size(); i++)
    {
        m_space_ptrs[i]->get_power_count();
        m_space_ptrs[i]->get_setpoints();
    }
}
```

Annex 5

Matlab code of the state space wall example

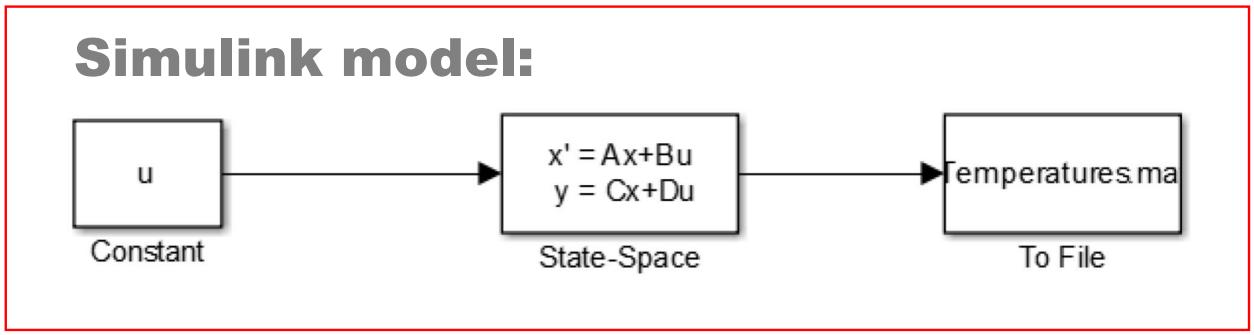
```
clear all

Ri  = 0.13 % [K/W] indoor air resistance
R12 = 3.00 % [K/W] resistance of the first construction layer
R23 = 0.056 % [K/W] resistance of the second construction layer
Re  = 0.04 % [K/W] indoor air resistance

C1  = 2400 % [J/K] heat capacity at the interior surface
C2  = 104400 % [J/K] heat capacity in between layers
C3  = 102000 % [J/K] heat capacity at the exterior surface


A   = [(-1/(Ri*C1) - 1/(R12*C1)) (1/(R12*C1)) 0;
       (1/(R12*C2)) (-1/(R12*C2) - 1/(R23*C2)) (1/(R23*C2));
       0 (1/(R23*C3)) ((-1/(R23*C3))+(-1/(Re*C3)))]
B   = [(1/(Ri*C1)) 0; 0 0; 0 (1/(Re*C3))]
C   = [1 0 0;0 1 0; 0 0 1]
D   = [0 0; 0 0; 0 0]
x0  = [5; 5; 5] %initial states
u   = [20; -10]

system1 = ss (A,B,C,D)
```



Annex 6

Input file of the resistance between two states example

```
#Day light savings time ON/OFF (resp. 1/0) (not yet implemented)
D,0

#Warm up days
U,0

#Time steps per hour
T,4

#Weather,   Start year, Start month,    Start day,  Start day name, Start hour, Finish year,    Finish month,   Finish day, Finish
hour,
E,      1985,       9,      1,      default,    1,      1985,       10,     1,      24

#Ground profile,Temperature
G,      50

#Materials, Material_ID,    Name,       Spec_Weight,    Spec_Heat,  Thermal_Conduc
M,      1,      Concrete,   2400.0,     850.0,      1.8
M,      2,      Isolatie,   60.0,       850.0,      0.04

#Constructions, Construct_ID,   mat_ID_1,   thickness_1,    ..........  mat_ID_n,   thickness_n
C,      1,      1,      100.0,              2,      50.0

#Spaces,    Space_ID,   Volume [m³],    Heating [W/m³], Cooling [W/m³]  Heat set point  Cool set point  Air changes/hour
S,      S_1,        27,     0.0,        0.0,        20.0,       22.0,       0.0

#Walls,     Wall_ID     Construction    Area [m²],  Orientation,    Space side one, Space side two
W,      W_1,        1,      9.0,        0,      G,      G
W,      W_2,        1,      9.0,        90,     G,      G
W,      W_3,        1,      9.0,        180,        G,      G
W,      W_4,        1,      9.0,        270,        G,      G

#Windows,   Window_ID,  Area [m²],  U_value [K/m²W] Cap./Area,  Orientation Space side one, Space side two


#Floors,    Floor_ID,   Construction,   Area,       Space side one, Space side two
F,      F_1,        1,      9.0,        G,      G
F,      F_2,        1,      9.0,        G,      G
```

Annex 7

Input file concrete box example

```
#Day light savings time ON/OFF (resp. 1/0) (not yet implemented)
D,0

#Warm up days
U,6

#Time steps per hour
T,4

#Weather,   Start year, Start month,    Start day,  Start day name, Start hour, Finish year,    Finish month,   Finish day, Finish
hour,
E,      1985,       9,      1,      default,    1,      1985,       10,     1,      24

#Ground profile,Temperature
G,      10

#Materials, Material_ID,    Name,       Spec_Weight,    Spec_Heat,  Thermal_Conduc
M,      1,      Concrete,   2400.0,     850.0,      1.8
M,      2,      Isolatie,   60.0,       850.0,      0.04

#Constructions, Construct_ID,   mat_ID_1,   thickness_1,    ..........  mat_ID_n,   thickness_n
C,      1,      1,      100.0,              2,      50.0

#Spaces,    Space_ID,   Volume [m³],    Heating [W/m³], Cooling [W/m³]  Heat set point  Cool set point  Air changes/hour
S,      1,      27,     0.0,        0.0,        20.0,       22.0,       0.0

#Walls,     Wall_ID     Construction    Area [m²],  Orientation,    Space side one, Space side two
W,      1,      1,      9.0,        0,      1,      E
W,      2,      1,      9.0,        90,     1,      E
W,      3,      1,      9.0,        180,        1,      E
W,      4,      1,      9.0,        270,        1,      E

#Floors,    Floor_ID,   Construction,   Area,       Space side one, Space side two
F,      1,      1,      9.0,        1,      E
F,      2,      1,      9.0,        1,      G
```

Annex 8

C++ code of the classes in the conformation

```
#ifndef VERTEX_H
#define VERTEX_H

#include <vector>


class Point;
class Line;
class Edge;
class Rectangle;
class Surface;
class Cuboid;
class Space;
class Vertex_Store;


class Vertex
{
private:
    std::vector<Point*> m_points;
    double m_x, m_y, m_z;
public:
    Vertex(double x, double y, double z);
    Vertex();
    ~Vertex();
    bool operator == (const Vertex& rhs);

    void add_point(Point* point_ptr);

    double get_x();
    double get_y();
    double get_z();

    Point* get_last_point_ptr();
};


class Point
{
private:
    // members are:
    Vertex* m_vertex; // this point is represented by one vertex

    // member of:
    std::vector<Edge*> m_edge_ptrs; // the edges this point belongs to (3 if cuboid)
    std::vector<Surface*> m_surface_ptrs; // the surfaces this point belongs to (3 if cuboid)
    std::vector<Space*> m_space_ptrs; // the spaces this point belongs to (1 in 3D-cases)
public:
    Point(Vertex* initial_vertex);
    ~Point();

    Vertex* get_vertex_ptr();

    void add_edge(Edge* edge_ptr);
    void add_surface(Surface* surface_ptr);
    void add_space(Space* space_ptr);
};

class Vector // direction of a line from base point vertex to the vertex to which the line
moves
{
private:
```

```
    double m_dx, m_dy, m_dz;
public:
    Vector(Vertex* base_pt, Vertex* move_pt);
    Vector(double dx, double dy, double dz);
    Vector();

    double get_dx();
    double get_dy();
    double get_dz();

    double calc_pt_product(const Vector& rhs);
    double calc_magnitude();
    Vector calc_cross_product(const Vector& rhs);
    bool same_direction(const Vector& rhs);
    bool is_zero();
    Vector change_direction();
};


class Line
{
private:
    std::vector<Edge*> m_edges; // this line belongs to a number of edges
    Vertex_Store* m_store_ptr; // pointer towards the vector in which this object of the
    Line class is stored
    Vertex* m_vertices[2];
    Vertex m_center_vertex;
    bool m_deletion;
public:
    Line(Vertex* one,Vertex* two, Vertex_Store* store_ptr);
    Line();
    ~Line();

    bool operator == (const Line& rhs);

    double get_length();

    bool check_vertex(Vertex* vertex_ptr);
    bool check_line_intersect(Line* line_ptr, Vertex* intersect_ptr);
    bool check_deletion();
    void tag_deletion();

    void add_edge(Edge* edge_ptr);
    void check_associated_members(Vertex* v_check);
    void split(Vertex* vertex_ptr);
    Vertex_Store* get_store_ptr();

    Vertex* get_vertex_ptr(int n);
    Vertex* get_center_vertex_ptr();
    Vertex* line_point_closest_to_vertex(Vertex* v_1);

    unsigned int get_edge_count();
    Edge* get_edge_ptr(unsigned int n);
    Edge* get_last_edge_ptr();
    Point* get_point_ptrs(int n);
};


class Edge
{
private:
    // members are:
```

```
    std::vector<Line*> m_lines; // this edge is represented by one or more lines (at least
    1, the initial line, see ctor)
    std::vector<Vertex*> m_vertices; // the vertices that represent this edge
    Line m_encasing_line;
    Point* m_points[2]; // this edge has 2 points

    // member of:
    std::vector<Surface*> m_surface_ptrs; // the surfaces this edge belongs to (2 if cuboid)
    std::vector<Space*> m_space_ptrs; // the spaces this edge belongs to (1 in 3D-cases)
public:
    Edge(Line* initial_line, Point* p_1, Point* p_2);
    ~Edge();

    void add_surface(Surface* surface_ptr);
    void add_space(Space* space_ptr);

    void add_line(Line* line_ptr);
    void delete_line(Line* line_ptr);

    unsigned int get_space_count();
    Space* get_space_ptr(unsigned int n);
    unsigned int get_surface_count();
    Surface* get_surface_ptr(unsigned int n);

    void check_vertex(Vertex* vertex_ptr); // checks if a vertex interferes with an edge
    that this object represents, if so appropriate action is taken and "true" is returned
    afterwards
    Line& get_encasing_line();
};


class Rectangle
{
private:
    std::vector<Surface*> m_surfaces;
    Vertex_Store* m_store_ptr; // pointer towards the vector in which this object of the
    Rectangle class is stored
    Line* m_lines[4];
    Vertex* m_vertices[4];
    Vector m_normal_vector;
    Vertex m_center_vertex;
    bool m_deletion;
public:
    Rectangle(Line* one, Line* two, Line* three, Line* four, Vertex_Store* store_ptr);
    Rectangle();
    ~Rectangle();

    bool operator == (const Rectangle& rhs);

    double get_area();

    bool check_vertex(Vertex* vertex_ptr);
    bool check_line_intersect(Line* line_ptr, Vertex* intersect_ptr);
    bool check_deletion();
    void tag_deletion();

    void add_surface(Surface* surface_ptr);
    void check_associated_members(Vertex* v_check);
    void split(Vertex* v_split);
    Vertex_Store* get_store_ptr();

    Line* get_line_ptr(int n);
```

```
    Vector* get_normal_ptr();
    Vertex* get_vertex_ptr(int n);
    Vertex* get_center_vertex_ptr();
    Vertex* surf_point_closest_vertex(Vertex* vertex_ptr);

    unsigned int get_surface_count();
    Surface* get_surface_ptr(unsigned int n);
    Surface* get_last_surface_ptr();
    Edge* get_edge_ptr(int n);
    Point* get_point_ptrs(int n);
};

class Surface
{
private:
    // members are:
    std::vector<Rectangle*> m_rectangles; // this surface is represented by one or more
    rectangles
    std::vector<Vertex*> m_vertices; // the vertices that represent this surface
    Rectangle m_encasing_rectangle;
    Edge* m_edges[4]; // this surface has 4 edges

    // member of:
    std::vector<Space*> m_space_ptrs; // the spaces that this surface belongs to (1 in
    3D-cases)
public:
    Surface(Rectangle* initial_rectangle, Edge* e_1, Edge* e_2, Edge* e_3, Edge* e_4);
    ~Surface();

    void add_space(Space* space_ptr);

    void add_rectangle(Rectangle*& rectangle_ptr);
    void delete_rectangle(Rectangle* rectangle_ptr);


    unsigned int get_space_count();
    Space* get_space_ptr(unsigned int i);
    Edge* get_edge_ptr(int n);

    void check_vertex(Vertex* vertex_ptr);
    Rectangle& get_encasing_rectangle();
};


class Cuboid
{
private:
    std::vector<Space*> m_spaces; // if .size() > 1 there are overlapping spaces.
    Vertex_Store* m_store_ptr;
    Rectangle* m_rectangles[6];
    Line* m_lines[12];
    Vertex* m_vertices[8];
    Vertex m_center_vertex;
    bool m_deletion;
public:
    Cuboid(Rectangle* one, Rectangle* two, Rectangle* three, Rectangle* four, Rectangle*
    five, Rectangle* six, Vertex_Store* store_ptr);
    Cuboid();
    ~Cuboid();

    bool operator == (const Cuboid& rhs);
```

```
    double get_volume();

    bool check_vertex(Vertex* vertex_ptr);
    bool check_deletion();
    void tag_deletion();

    void add_space(Space* space_ptr);
    void check_associated_members(Vertex* v_check);
    void split(Vertex* v_split);
    Vertex_Store* get_store_ptr();

    unsigned int get_space_count();
    Vertex* get_center_vertex_ptr();
    Vertex* get_vertex_ptr(int n);
    Vertex* get_max_vertex();
    Vertex* get_min_vertex();
    Line* get_line_ptr(int n);
    Rectangle* get_rectangle_ptr(int n);
    Space* get_last_space_ptr();
    Space* get_space_ptr (unsigned int n);
    Surface* get_surface_ptrs(int n);
    unsigned int get_edge_count();
    Edge* get_edge_ptrs(int n);
    Point* get_point_ptrs(int n);
};

class Space
{
private:
    // members are:
    std::vector<Cuboid*> m_cuboids; // this space is represented by one or more cuboid
    std::vector<Vertex*> m_vertices;
    Cuboid m_encasing_cuboid;
    Surface* m_surfaces[6]; // this space has 6 surfaces
    Edge* m_edges[12]; // this space has 12 edges
    Point* m_points[8]; // this space has 8 points

    int m_space_ID;
public:
    Space(int room_ID, Cuboid* initial_cuboid, Surface* s_1, Surface* s_2, Surface* s_3,
    Surface* s_4, Surface* s_5, Surface* s_6);
    ~Space();

    int get_ID();
    void add_cuboid(Cuboid* cuboid_ptr);
    void delete_cuboid(Cuboid* cuboid_ptr);

    Edge* get_edge_ptr(int n);
    Surface* get_surface_ptr(int n);

    void check_vertex(Vertex* vertex_ptr);
    Cuboid& get_encasing_cuboid();
};


class Vertex_Store
{
protected:
    std::vector<Vertex*> m_vertices;
    std::vector<Line*> m_lines;
    std::vector<Rectangle*> m_rectangles;
    std::vector<Cuboid*> m_cubes;
```

```
public:
    Vertex_Store();
    ~Vertex_Store();

    Vertex* add_vertex(double x, double y, double z);
    Line* add_line(Vertex* one, Vertex* two);
    Rectangle* add_rectangle(Line* one, Line* two, Line* three, Line* four);
    Cuboid* add_cuboid(Rectangle* one, Rectangle* two, Rectangle* three, Rectangle* four,
    Rectangle* five, Rectangle* six);

    void delete_line(Line* l_d);
    void delete_rectangle(Rectangle* r_d);
    void delete_cuboid(Cuboid* c_d);
};
#endif // VERTEX_H
```

```cpp
#include "../include/Vertex.h"
#include <iostream>
#include <cmath>

Vertex::Vertex(double x, double y, double z)
{
    m_x = x;
    m_y = y;
    m_z = z;
    //ctor
}

Vertex::Vertex()
{
    //ctor
}

Vertex::~Vertex()
{
    //dtor
}

bool Vertex::operator == (const Vertex& rhs)
{
    if (this == &rhs)
    {
        return true;
    }
    else if ((m_x == rhs.m_x) && (m_y == rhs.m_y) && (m_z == rhs.m_z))
    {
        return true;
    }
    else
    {
        return false;
    }
}

void Vertex::add_point(Point* point_ptr)
{
    bool ptr_found = false;
    for (unsigned int i = 0; i < m_points.size(); i++)
    {
        if (point_ptr == m_points[i])
        {
            ptr_found = true;
        }
    }
    if (!ptr_found)
    {
        m_points.push_back(point_ptr);
    }
}

double Vertex::get_x()
{
    return m_x;
}

double Vertex::get_y()
{
    return m_y;
```

```
}

double Vertex::get_z()
{
    return m_z;
}


Point* Vertex::get_last_point_ptr()
{
    return m_points.back();
}









Point::Point(Vertex* initial_vertex)
{
    m_vertex = initial_vertex;
    //ctor
}

Point::~Point()
{
    //dtor
}

Vertex* Point::get_vertex_ptr()
{
    return m_vertex;
}

void Point::add_edge(Edge* edge_ptr)
{
    m_edge_ptrs.push_back(edge_ptr);
}

void Point::add_surface(Surface* surface_ptr)
{
    m_surface_ptrs.push_back(surface_ptr);
}

void Point::add_space(Space* space_ptr)
{
    m_space_ptrs.push_back(space_ptr);
    if (m_space_ptrs.size() > 1)
    {
        std::cout << "Error, point assigned to too many spaces (only 1 possible), exiting.."
        << std::endl;
        exit(1);
    }
}
```

```
Vector::Vector(Vertex* base_pt, Vertex* move_pt)
{
    m_dx = (move_pt->get_x() - base_pt->get_x());
    m_dy = (move_pt->get_y() - base_pt->get_y());
    m_dz = (move_pt->get_z() - base_pt->get_z());
}

Vector::Vector(double dx, double dy, double dz)
{
    m_dx = dx;
    m_dy = dy;
    m_dz = dz;
}

Vector::Vector()
{
    //ctor
}

double Vector::get_dx()
{
    return m_dx;
}

double Vector::get_dy()
{
    return m_dy;
}

double Vector::get_dz()
{
    return m_dz;
}

double Vector::calc_pt_product(const Vector& rhs)
{
    double sum = 0;
    sum += m_dx*rhs.m_dx;
    sum += m_dy*rhs.m_dy;
    sum += m_dz*rhs.m_dz;
    return sum;
}

double Vector::calc_magnitude()
{
    return sqrt(pow(m_dx,2) + pow(m_dy,2) + pow(m_dz,2));
}

Vector Vector::calc_cross_product(const Vector& rhs)
{
    return Vector(m_dy*rhs.m_dz - m_dz*rhs.m_dy, m_dz*rhs.m_dx - m_dx*rhs.m_dz,
    m_dx*rhs.m_dy - m_dy*rhs.m_dx);
}

bool Vector::same_direction(const Vector& rhs)
{
    if (((m_dx >= 0) ^ (rhs.m_dx < 0)) &&
        ((m_dy >= 0) ^ (rhs.m_dy < 0)) &&
        ((m_dz >= 0) ^ (rhs.m_dz < 0)))
    {
        return true;
```

```
    }
    else
    {
        return false;
    }
}

bool Vector::is_zero()
{
    if (m_dx == 0 && (m_dy == 0 && m_dz == 0))
    {
        return true;
    }
    else
    {
        return false;
    }
}
Vector Vector::change_direction()
{
    Vector v_temp;
    v_temp.m_dx = -m_dx;
    v_temp.m_dy = -m_dy;
    v_temp.m_dz = -m_dz;
    return v_temp;
}
```

```cpp
#include "../include/Vertex.h"
#include <iostream>
#include <cmath>
#include <algorithm>
#include <cstdlib>

Line::Line(Vertex* one, Vertex* two, Vertex_Store* store_ptr)
{
    m_store_ptr = store_ptr;
    m_deletion = false;
    m_vertices[0] = one;
    m_vertices[1] = two;

    // calculate the center vertex
    double sum_x = 0, sum_y = 0, sum_z = 0;
    for (int i = 0; i < 2; i++)
    {
        sum_x += m_vertices[i]->get_x();
        sum_y += m_vertices[i]->get_y();
        sum_z += m_vertices[i]->get_z();
    }
    m_center_vertex = Vertex(sum_x/2.0, sum_y/2.0, sum_x/2.0);
    //ctor
}

Line::Line()
{
    //ctor
}

Line::~Line()
{
    //dtor
}

bool Line::operator == (const Line& rhs)
{
    if (this == &rhs)
    {
        return true;
    }
    else if ((*m_vertices[0] == *rhs.m_vertices[0] && *m_vertices[1] == *rhs.m_vertices[1]) ||
        (*m_vertices[0] == *rhs.m_vertices[1] && *m_vertices[1] == *rhs.m_vertices[0]))
    {
        return true;
    }
    else
    {
        return false;
    }
}

double Line::get_length()
{
    Vector vct_1(m_vertices[0], m_vertices[1]);
    return vct_1.calc_magnitude();
}

bool Line::check_vertex(Vertex* vertex_ptr)
{
    for (int i = 0; i < 2; i++)
    {
```

```
        if (*vertex_ptr == *m_vertices[i])
        {
            return false;
        }
    }

    Vector vct_1(m_vertices[0], m_vertices[1]);
    Vector vct_2(m_vertices[0], vertex_ptr);
    Vector vct_3(m_vertices[1], vertex_ptr);

    if (!vct_1.calc_cross_product(vct_2).is_zero()) // check if both vectors are parallel,
    if so they are collinear
    {
        return false;
    }

    int true_count_tot = 0;

    if ((vct_2.get_dx() >  0) && (vct_3.get_dx() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_2.get_dx() < 0) && (vct_3.get_dx() < 0))
    {
        true_count_tot++;
    }

    if ((vct_2.get_dy() > 0) && (vct_3.get_dy() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_2.get_dy() < 0) && (vct_3.get_dy() < 0))
    {
        true_count_tot++;
    }

    if ((vct_2.get_dz() >  0) && (vct_3.get_dz() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_2.get_dz() < 0) && (vct_3.get_dz() < 0))
    {
        true_count_tot++;
    }

    if (true_count_tot > 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}

bool Line::check_line_intersect(Line* line_ptr, Vertex* intersect_ptr)
{
    Vector v_1(m_vertices[0], m_vertices[1]);
    Vector v_2(line_ptr->get_vertex_ptr(0), line_ptr->get_vertex_ptr(1));
    Vector v_3(m_vertices[0], line_ptr->get_vertex_ptr(0));

    if ((v_1.calc_cross_product(v_2).is_zero()) && (v_1.calc_cross_product(v_3).is_zero()))
```

```
    // two lines are colinear
    {
        return false;
    }
    else if ((v_1.calc_cross_product(v_2).is_zero())) // two lines are parallel
    {
        return false;
    }
    else if (v_3.calc_pt_product(v_1.calc_cross_product(v_2)) != 0) // two lines are skew
    and thus not coplanar
    {
        return false;
    }
    // if none of the above are true, then there exists a point where vertices v_1 and v_2
    intersect (given the points entered above)
    double s = (v_2.calc_cross_product(v_3).calc_magnitude()) /
               (v_2.calc_cross_product(v_1).calc_magnitude());
    double x = m_vertices[0]->get_x() + s * v_1.get_dx(),
           y = m_vertices[0]->get_y() + s * v_1.get_dy(),
           z = m_vertices[0]->get_z() + s * v_1.get_dz();

    *intersect_ptr = Vertex(x, y, z);

    // now check if the point lies on both lines
    if (line_ptr->check_vertex(intersect_ptr) && this->check_vertex(intersect_ptr))
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool Line::check_deletion()
{
    return m_deletion;
}

void Line::tag_deletion()
{
    m_deletion = true;
}

void Line::add_edge(Edge* edge_ptr)
{
    if (std::find(m_edges.begin(), m_edges.end(), edge_ptr) == m_edges.end())
    {
        m_edges.push_back(edge_ptr);
    }
}

void Line::check_associated_members(Vertex* v_check)
{
    for (unsigned int i = 0; i < this->get_edge_count(); i++)
    {
        this->get_edge_ptr(i)->check_vertex(v_check);
        for (unsigned int j = 0; j <  this->get_edge_ptr(i)->get_surface_count(); j++)
        {
            this->get_edge_ptr(i)->get_surface_ptr(j)->check_vertex(v_check);
        }
```

```
        for (unsigned int j = 0; j < this->get_edge_ptr(i)->get_space_count(); j++)
        {
            this->get_edge_ptr(i)->get_space_ptr(j)->check_vertex(v_check);
        }
    }
}

void Line::split(Vertex* v_split)
{
    this->tag_deletion();

    Line* l_1 = m_store_ptr->add_line(m_vertices[0], v_split);
    Line* l_2 = m_store_ptr->add_line(m_vertices[1], v_split);

    for (unsigned int i = 0; i < m_edges.size(); i++)
    {
        l_1->add_edge(m_edges[i]); // fill or update m_edges in l_1 with the edge pointers
        contained in this object
        l_2->add_edge(m_edges[i]); // fill or update m_edges in l_2 with the edge pointers
        contained in this object
        m_edges[i]->add_line(l_1); // update all edges that this object is referenced to
        with the new line pointed to by l_1
        m_edges[i]->add_line(l_2); // update all edges that this object is referenced to
        with the new line pointed to by l_2
        m_edges[i]->delete_line(this); // remove this line from all associated edges, as
        these have now been updated
    }

    // check the edges, surfaces and spaces associated to this object with the new vertices:
    v_split
    this->check_associated_members(v_split);

    m_edges.clear(); // clear the vector, this line object now does no longer belong to any
    edge
}

Vertex_Store* Line::get_store_ptr()
{
    return m_store_ptr;
}

Vertex* Line::get_vertex_ptr(int n)
{
    return m_vertices[n];
}

Vertex* Line::get_center_vertex_ptr()
{
    return &m_center_vertex;
}

Vertex* Line::line_point_closest_to_vertex(Vertex* check_vertex)
{
    Vector v_1(m_vertices[0], m_vertices[1]); // vector of this line
    Vector v_3(m_vertices[0], check_vertex); // vector from a point on this line to vertex_ptr
    Vector v_2 = v_1.calc_cross_product(v_3); // v_2 is perpendicular to both v_1 and v_3
    (normal of the plane containing all points)
    v_2 = v_1.calc_cross_product(v_2); // v_2 is now coplanar with the plane containing all
    3 vertexes and is perpendicular to v_1

    // now the intersection point of v_1 and v_2 is searched
    double s = (v_2.calc_cross_product(v_3).calc_magnitude()) /
```

```
                (v_2.calc_cross_product(v_1).calc_magnitude());

    double x = m_vertices[0]->get_x() + s * v_1.get_dx(),
           y = m_vertices[0]->get_y() + s * v_1.get_dy(),
           z = m_vertices[0]->get_z() + s * v_1.get_dz();

    Vertex* intersect_vertex = m_store_ptr->add_vertex(x, y, z);

    return intersect_vertex;
}

unsigned int Line::get_edge_count()
{
    return m_edges.size();
}

Edge* Line::get_edge_ptr(unsigned int n)
{
    return m_edges[n];
}

Edge* Line::get_last_edge_ptr()
{
    return m_edges.back();
}

Point* Line::get_point_ptrs(int n)
{
    return m_vertices[n]->get_last_point_ptr();
}
```

```
Edge::Edge(Line* initial_line, Point* p_1, Point* p_2)
{
    m_lines.push_back(initial_line);
    m_points[0] = p_1;
    m_points[1] = p_2;
    for (int i = 0; i < 2; i++)
    {
        m_vertices.push_back(initial_line->get_vertex_ptr(i));
    }
    m_encasing_line = Line(m_points[0]->get_vertex_ptr(),
                           m_points[1]->get_vertex_ptr(),
                           nullptr);
}

Edge::~Edge()
{
    //dtor
}

void Edge::add_surface(Surface* surface_ptr)
{
    m_surface_ptrs.push_back(surface_ptr);
```

```
    if (m_space_ptrs.size() > 2)
    {
        std::cout << "Error, edge assigned to too many surfaces (only 2 possible),
        exiting.." << std::endl;
        exit(1);
    }
}

void Edge::add_space(Space* space_ptr)
{
    m_space_ptrs.push_back(space_ptr);
    if (m_space_ptrs.size() > 1)
    {
        std::cout << "Error, edge assigned to too many spaces (only 1 possible), exiting.."
        << std::endl;
        exit(1);
    }
}

void Edge::add_line(Line* line_ptr)
{
    if (std::find(m_lines.begin(), m_lines.end(), line_ptr) == m_lines.end())
    {
        m_lines.push_back(line_ptr);
    }
}

void Edge::delete_line(Line* line_ptr)
{
    m_lines.erase( std::remove(m_lines.begin(), m_lines.end(), line_ptr), m_lines.end());
}

unsigned int Edge::get_space_count()
{
    return m_space_ptrs.size();
}

Space* Edge::get_space_ptr(unsigned int n)
{
    return m_space_ptrs[n]; // returns the last element of this vector, which cannot be
    larger than 1 element and must be larger than 0 elements
}

unsigned int Edge::get_surface_count()
{
    return m_surface_ptrs.size();
}

Surface* Edge::get_surface_ptr(unsigned int n)
{
    return m_surface_ptrs[n];
}

void Edge::check_vertex(Vertex* vertex_ptr) // checks if a vertex interferes with an edge
that this object represents, if so appropriate action is taken
{
    if (!m_encasing_line.check_vertex(vertex_ptr)) // check if the vertex is located in or
    on the space's encasing cuboid
    {
        return;
    }
    if (std::find(m_vertices.begin(), m_vertices.end(), vertex_ptr) == m_vertices.end()) //
```

```
    if the vertex is not part of the edge yet, then add it to the edge and continue with the
    rest of the function
    {
        m_vertices.push_back(vertex_ptr);
    }
    else // if it already is added to the edge, then no further action is required and the
    function is ended
    {
        return;
    }

    for (unsigned int i = 0; i < m_lines.size(); i++) // find the lines in which or on which
    the vertex is located
    {
        if (m_lines[i]->check_vertex(vertex_ptr))
        {
            m_lines[i]->split(vertex_ptr); // split this line in 2 lines
        }
    }
}

Line& Edge::get_encasing_line()
{
    return m_encasing_line;
}
```

```cpp
#include "../include/Vertex.h"
#include <iostream>
#include <cstdlib>
#include <algorithm>
#include <cstdlib>

Rectangle::Rectangle(Line* l_one, Line* l_two, Line* l_three, Line* l_four, Vertex_Store*
store_ptr)
{
    m_store_ptr = store_ptr;
    m_deletion = false;

    std::fill_n(m_lines, 4, nullptr);
    m_lines[0] = l_one;

    Line* temp_container[3];
    temp_container[0] = l_two;
    temp_container[1] = l_three;
    temp_container[2] = l_four;

    // find the lines parallel and perpendicular to m_lines[0]
    Vector vct_1(m_lines[0]->get_vertex_ptr(0), m_lines[0]->get_vertex_ptr(1));
    for (int i = 0; i < 3; i++)
    {
        Line* temp_ptr = temp_container[i];
        Vector vct_2(temp_ptr->get_vertex_ptr(0), temp_ptr->get_vertex_ptr(1));
        if (vct_1.calc_cross_product(vct_2).is_zero())
        {
            m_lines[2] = temp_ptr;
        }
        else if (vct_1.calc_pt_product(vct_2) == 0)
        {
            if ((*m_lines[0]->get_vertex_ptr(0) == *temp_ptr->get_vertex_ptr(0)) ||
                (*m_lines[0]->get_vertex_ptr(0) == *temp_ptr->get_vertex_ptr(1)))
            {
                m_lines[1] = temp_ptr;
            }
            else if ((*m_lines[0]->get_vertex_ptr(1) == *temp_ptr->get_vertex_ptr(0)) ||
                     (*m_lines[0]->get_vertex_ptr(1) == *temp_ptr->get_vertex_ptr(1)))
            {
                m_lines[3] = temp_ptr;
            }
            else
            {
                std::cout << "Error in initialization of rectangle(a): lines are not
                adjoint, exiting.." << std::endl;
                exit(1);
            }
        }
        else
        {
            std::cout << "Error in initialization of rectangle: lines are not perpendicular,
            exiting.." << std::endl;
            exit(1);
        }
    }

    for (int i = 0; i < 4; i++)
    {
        if (m_lines[i] == nullptr)
        {
            std::cout << "Error, not all lines were initialised correctly, exiting.." <<
```

```
            std::endl;
            exit(1);
        }
    }

    // all vertices should be all vertices on m_lines[0] and m_lines[3] (the two parallel
    vertices)
    m_vertices[0] = m_lines[0]->get_vertex_ptr(0);
    m_vertices[1] = m_lines[0]->get_vertex_ptr(1);
    m_vertices[2] = m_lines[2]->get_vertex_ptr(0);
    m_vertices[3] = m_lines[2]->get_vertex_ptr(1);

    // check if m_lines[3] has its vertices in common with m_lines[2] and m_lines[4]
    if ((*m_lines[2]->get_vertex_ptr(0) == *m_lines[1]->get_vertex_ptr(0)) ||
        (*m_lines[2]->get_vertex_ptr(0) == *m_lines[1]->get_vertex_ptr(1)))
    {
        if (!((*m_lines[2]->get_vertex_ptr(1) == *m_lines[3]->get_vertex_ptr(0)) ||
              (*m_lines[2]->get_vertex_ptr(1) == *m_lines[3]->get_vertex_ptr(1))))
        {
            std::cout << "Error in initialization of rectangle(b): lines are not adjoint,
            exiting.." << std::endl;
            exit(1);
        }
    }
    else if ((*m_lines[2]->get_vertex_ptr(0) == *m_lines[3]->get_vertex_ptr(0)) ||
             (*m_lines[2]->get_vertex_ptr(0) == *m_lines[3]->get_vertex_ptr(1)))
    {
        if (!((*m_lines[2]->get_vertex_ptr(1) == *m_lines[1]->get_vertex_ptr(0)) ||
              (*m_lines[2]->get_vertex_ptr(1) == *m_lines[1]->get_vertex_ptr(1))))
        {
            std::cout << "Error in initialization of rectangle(c): lines are not adjoint,
            exiting.." << std::endl;
            exit(1);
        }
    }
    else
    {
        std::cout << "Error in initialization of rectangle(d): lines are not adjoint,
        exiting.." << std::endl;
        exit(1);
    }

    // compute the normal vector
    Vector vct_2(m_lines[1]->get_vertex_ptr(0), m_lines[1]->get_vertex_ptr(1));
    m_normal_vector = vct_1.calc_cross_product(vct_2);

    // calculate the center vertex
    double sum_x = 0, sum_y = 0, sum_z = 0;
    for (int i = 0; i < 4; i++)
    {
        sum_x += m_vertices[i]->get_x();
        sum_y += m_vertices[i]->get_y();
        sum_z += m_vertices[i]->get_z();
    }
    m_center_vertex = Vertex(sum_x/4.0, sum_y/4.0, sum_x/4.0);
    //ctor
}

Rectangle::Rectangle()
{
    //ctor
}
```

```
Rectangle::~Rectangle()
{
    //dtor
}

bool Rectangle::operator == (const Rectangle& rhs)
{
    if (this == &rhs)
    {
        return true;
    }
    else
    {
        for (int i = 0; i < 4; i++)
        {
            if (m_lines[0] == rhs.m_lines[i])
            {
                for (int j = 0; j < 4; j++)
                {
                    if ((j != i) && (m_lines [1] == rhs.m_lines[j]))
                    {
                        return true; // if two lines of a rectangle surface are coincident,
                        then the entire rectangle is coincident
                    }
                }
            }
        }
        return false;
    }
}

double Rectangle::get_area()
{
    Vector vct_1(m_lines[0]->get_vertex_ptr(0), m_lines[0]->get_vertex_ptr(1));
    Vector vct_2;

    for (int i = 1; i < 4; i++)
    {
        vct_2 = Vector(m_lines[i]->get_vertex_ptr(0), m_lines[i]->get_vertex_ptr(1));
        if (vct_1.calc_pt_product(vct_2) == 0) // if perpendicular
        {
            return (m_lines[0]->get_length() * m_lines[i]->get_length()); // if they are
            perpendicular return their product as their area
        }
    }
    std::cout << "Error in calculating Rectangle area, exiting..." << std::endl;
    exit(1);
}

bool Rectangle::check_vertex(Vertex* vertex_ptr)
{
    for (int i = 0; i < 4; i++) // check all vertices of the rectangle
    {
        if (*vertex_ptr == *m_vertices[i]) // if vertex_ptr is part of the rectangle already
        {
            return false; // then return false
        }
    }

    Vector vct_1(m_vertices[0], vertex_ptr);
    Vector vct_2(m_vertices[1], vertex_ptr);
```

```
    Vector vct_3(m_vertices[2], vertex_ptr);

    if (vct_1.calc_pt_product(m_normal_vector) != 0)
    {
        return false;
    }

    int true_count_tot = 0;

    if ((vct_1.get_dx() > 0) && (vct_2.get_dx() > 0) && (vct_3.get_dx() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_1.get_dx() < 0) && (vct_2.get_dx() < 0) && (vct_3.get_dx() < 0))
    {
        true_count_tot++;
    }

    if ((vct_1.get_dy() > 0) && (vct_2.get_dy() > 0) && (vct_3.get_dy() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_1.get_dy() < 0) && (vct_2.get_dy() < 0) && (vct_3.get_dy() < 0))
    {
        true_count_tot++;
    }

    if ((vct_1.get_dz() > 0) && (vct_2.get_dz() > 0) && (vct_3.get_dz() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_1.get_dz() < 0) && (vct_2.get_dz() < 0) && (vct_3.get_dz() < 0))
    {
        true_count_tot++;
    }

    if (true_count_tot > 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}

bool Rectangle::check_line_intersect(Line* line_ptr, Vertex* intersect_ptr)
{
    Vector vct_1(line_ptr->get_vertex_ptr(0),line_ptr->get_vertex_ptr(1)); // directional
    vector of the line from vertex A to vertex B
    Vector vct_2(line_ptr->get_vertex_ptr(0),m_vertices[0]); // directional vector from
    vertex A to vertex C (which is a point defining the plane, together with the normal
    vector of the rectangle)

    if (vct_1.calc_pt_product(m_normal_vector) == 0) // check if the line is orthogonal to
    the plane normal, if so, the line is parallel to the plane or coplanar
    {
        return false;
    }
    else // if not, then the line or its extension intersects the plane at a point
    {
        // determine the scalar for the vector of the line where the line intersects the
```

```
        plane i.e.: Intersection point = A + s.v_1
        double s = (m_normal_vector.calc_pt_product(vct_2) /
                    m_normal_vector.calc_pt_product(vct_1));

        //compute the point with scalar s
        double x = line_ptr->get_vertex_ptr(0)->get_x() + s * vct_1.get_dx(),
               y = line_ptr->get_vertex_ptr(0)->get_y() + s * vct_1.get_dy(),
               z = line_ptr->get_vertex_ptr(0)->get_z() + s * vct_1.get_dz();

        // use the Vertex pointer to pass the intersection point
        *intersect_ptr = Vertex(x, y, z);

        if (this->check_vertex(intersect_ptr) && line_ptr->check_vertex(intersect_ptr)) //
        if the point lies within the rectangle and on the line, then the line intersects the
        rectangle
        {
            return true;
        }
        else // if not the intersection point of the line and the plane of the rectangle
        lies outside the rectangle or outside the line (i.e. s < 0 or s > 1
        {
            return false;
        }
    }
}

bool Rectangle::check_deletion()
{
    return m_deletion;
}

void Rectangle::tag_deletion()
{
    m_deletion = true;
}

void Rectangle::add_surface(Surface* surface_ptr)
{
    if (std::find(m_surfaces.begin(), m_surfaces.end(), surface_ptr) == m_surfaces.end())
    {
        m_surfaces.push_back(surface_ptr);
    }
}

void Rectangle::check_associated_members(Vertex* v_check)
{
    for (unsigned int i = 0; i < this->get_surface_count(); i++)
    {
        for (int j = 0; j < 4; j++)
        {
            this->get_surface_ptr(i)->get_edge_ptr(j)->check_vertex(v_check);
        }
        this->get_surface_ptr(i)->check_vertex(v_check);

        for (unsigned int j = 0; j < this->get_surface_ptr(i)->get_space_count(); j++)
        {
            this->get_surface_ptr(i)->get_space_ptr(j)->check_vertex(v_check);
        }
    }
}

void Rectangle::split(Vertex* v_split)
```

```
{
    this->tag_deletion();

    for (int i = 0; i < 4; i++) // if the vertex lies on one of the rectangle's lines, then
    the rectangle is to be split into two new rectangles
    {
        if (m_lines[i]->check_vertex(v_split))
        {
            m_lines[i]->tag_deletion();
            Vertex* v_1 = nullptr, * v_2 = nullptr, * v_3 = nullptr, * v_4 = nullptr, * v_5
            = nullptr; // initialise these to nullptrs to suppress a warning about the use
            of uninitialized variables
            Line* l_1 = nullptr, * l_2 = nullptr, * l_3, * l_4, * l_5, * l_6, * l_7;
            Rectangle * r_1, * r_2;

            v_1 = m_lines[i]->get_vertex_ptr(0);
            v_2 = m_lines[i]->get_vertex_ptr(1);

            Vector vct_1(m_lines[i]->get_vertex_ptr(0), v_split);
            Vector vct_2;

            // find v_1 and v_2
            for (int j = 0; j < 4; j++) // compare properties of m_line[0] with those of
            other lines of the rectangle
            {
                if (!(*m_lines[i] == *m_lines[j]))
                {
                    if ((*m_lines[j]->get_vertex_ptr(0) == *v_1) ||
                    (*m_lines[j]->get_vertex_ptr(1) == *v_1))
                    {
                        l_1 = m_lines[j];
                        if (*m_lines[j]->get_vertex_ptr(0) == *v_1)
                        {
                            v_3 = m_lines[j]->get_vertex_ptr(1);
                        }
                        else
                        {
                            v_3 = m_lines[j]->get_vertex_ptr(0);
                        }
                    }
                    else if ((*m_lines[j]->get_vertex_ptr(0) == *v_2) ||
                    (*m_lines[j]->get_vertex_ptr(1) == *v_2))
                    {
                        l_2 = m_lines[j];
                        if (*m_lines[j]->get_vertex_ptr(0) == *v_2)
                        {
                            v_4 = m_lines[j]->get_vertex_ptr(1);
                        }
                        else
                        {
                            v_4 = m_lines[j]->get_vertex_ptr(0);
                        }
                    }
                    else
                    {
                        m_lines[j]->tag_deletion();
                        v_5 = m_lines[j]->line_point_closest_to_vertex(v_split);
                    }
                }
            }

            if ((v_1 == nullptr) || (v_2 == nullptr) || (v_3 == nullptr) || (v_4 == nullptr)
```

```
            || (v_5 == nullptr))
            {
                std::cout << "Error, not all vertices were initialized during the splitting
                of a rectangle(a), exiting..." << std::endl;
                exit(1);
            }

            if ((l_1 == nullptr) || (l_2 == nullptr))
            {
                std::cout << "Error, not all lines were initialized during the splitting of
                a rectangle (a), exiting..." << std::endl;
                exit(1);
            }

            l_3 = m_store_ptr->add_line(v_5, v_split);
            l_4 = m_store_ptr->add_line(v_3, v_5);
            l_5 = m_store_ptr->add_line(v_4, v_5);
            l_6 = m_store_ptr->add_line(v_1, v_split);
            l_7 = m_store_ptr->add_line(v_2, v_split);

            r_1 = m_store_ptr->add_rectangle(l_1, l_3, l_4, l_6);
            r_2 = m_store_ptr->add_rectangle(l_2, l_3, l_5, l_7);

            for (unsigned int i = 0; i < m_surfaces.size(); i++)
            {
                r_1->add_surface(m_surfaces[i]); // fill or update m_surfaces in r_1 with
                the surface pointers contained in this object
                r_2->add_surface(m_surfaces[i]); // fill or update m_surfaces in r_2 with
                the surface pointers contained in this object
                m_surfaces[i]->add_rectangle(r_1); // update all surfaces that this object
                is referenced to with the new rectangle pointed to by r_1
                m_surfaces[i]->add_rectangle(r_2); // update all surfaces that this object
                is referenced to with the new rectangle pointed to by r_2
                m_surfaces[i]->delete_rectangle(this); // remove this rectangle from all
                associated surfaces, as these have now been updated
            }

            // check the edges, surfaces and spaces associated to this object with the new
            vertices: v_5 and v_split
            this->check_associated_members(v_5);
            this->check_associated_members(v_split);

            m_surfaces.clear(); // clear the vector, this rectangle object now does no
            longer belong to any surface
            return;
        }
    }
    // if none of the above applies, then the vertex lies on the rectangle, and the
    rectangle is to be split into four new rectangles:
    for (int i = 0; i < 4; i++) // tag all lines for deletion
    {
        m_lines[i]->tag_deletion();
    }

    // initialize the vertices, lines and rectangles required to define the four new
    rectangles
    Vertex* v_1 = nullptr, * v_2 = nullptr, * v_3 = nullptr, * v_4 = nullptr,
          * v_5 = nullptr, * v_6 = nullptr, * v_7 = nullptr, * v_8 = nullptr; // initialise
          these to nullptrs to suppress a warning about the use of uninitialized variables
            Line* l_1, * l_2, * l_3, * l_4, * l_5, * l_6, * l_7, * l_8, * l_9, * l_10, *
            l_11, * l_12;
            Rectangle * r_1, * r_2, * r_3, * r_4;
```

```
v_1 = m_lines[0]->get_vertex_ptr(0);
v_2 = m_lines[0]->get_vertex_ptr(1);
v_5 = m_lines[0]->line_point_closest_to_vertex(v_split);


for (int i = 1; i < 4; i++) // compare properties of m_line[0] with those of other lines
of the rectangle
{
    if ((*m_lines[i]->get_vertex_ptr(0) == *v_1) || (*m_lines[i]->get_vertex_ptr(1) ==
    *v_1))
    {
        v_7 = m_lines[i]->line_point_closest_to_vertex(v_split);
        if (*m_lines[i]->get_vertex_ptr(0) == *v_1)
        {
            v_3 = m_lines[i]->get_vertex_ptr(1);
        }
        else
        {
            v_3 = m_lines[i]->get_vertex_ptr(0);
        }
    }
    else if ((*m_lines[i]->get_vertex_ptr(0) == *v_2) || (*m_lines[i]->get_vertex_ptr(1)
    == *v_2))
    {
        v_8 = m_lines[i]->line_point_closest_to_vertex(v_split);
        if (*m_lines[i]->get_vertex_ptr(0) == *v_2)
        {
            v_4 = m_lines[i]->get_vertex_ptr(1);
        }
        else
        {
            v_4 = m_lines[i]->get_vertex_ptr(0);
        }
    }
    else
    {
        v_6 = m_lines[i]->line_point_closest_to_vertex(v_split);
    }
}

if ((v_1 == nullptr) || (v_2 == nullptr) || (v_3 == nullptr) || (v_4 == nullptr) ||
    (v_5 == nullptr) || (v_6 == nullptr) || (v_7 == nullptr) || (v_8 == nullptr))
{
    std::cout << "Error, not all vertices were initialized during the splitting of a
    rectangle (b), exiting..." << std::endl;
    exit(1);
}

l_1  = m_store_ptr->add_line(v_1, v_7);
l_2  = m_store_ptr->add_line(v_7, v_3);
l_3  = m_store_ptr->add_line(v_1, v_5);
l_4  = m_store_ptr->add_line(v_7, v_split);
l_5  = m_store_ptr->add_line(v_3, v_6);
l_6  = m_store_ptr->add_line(v_5, v_split);
l_7  = m_store_ptr->add_line(v_6, v_split);
l_8  = m_store_ptr->add_line(v_5, v_2);
l_9  = m_store_ptr->add_line(v_8, v_split);
l_10 = m_store_ptr->add_line(v_6, v_4);
l_11 = m_store_ptr->add_line(v_2, v_8);
l_12 = m_store_ptr->add_line(v_8, v_4);
```

```
    r_1 = m_store_ptr->add_rectangle(l_1,  l_3,  l_4,  l_6 );
    r_2 = m_store_ptr->add_rectangle(l_2,  l_4,  l_5,  l_7 );
    r_3 = m_store_ptr->add_rectangle(l_6,  l_8,  l_9,  l_11);
    r_4 = m_store_ptr->add_rectangle(l_7,  l_9,  l_10, l_12);

    for (unsigned int i = 0; i < m_surfaces.size(); i++)
    {
        r_1->add_surface(m_surfaces[i]); // fill or update m_surfaces in r_1 with the
        surface pointers contained in this object
        r_2->add_surface(m_surfaces[i]); // fill or update m_surfaces in r_2 with the
        surface pointers contained in this object
        r_3->add_surface(m_surfaces[i]); // fill or update m_surfaces in r_3 with the
        surface pointers contained in this object
        r_4->add_surface(m_surfaces[i]); // fill or update m_surfaces in r_4 with the
        surface pointers contained in this object
        m_surfaces[i]->add_rectangle(r_1); // update all surfaces that this object is
        referenced to with the new rectangle pointed to by r_1
        m_surfaces[i]->add_rectangle(r_2); // update all surfaces that this object is
        referenced to with the new rectangle pointed to by r_2
        m_surfaces[i]->add_rectangle(r_3); // update all surfaces that this object is
        referenced to with the new rectangle pointed to by r_3
        m_surfaces[i]->add_rectangle(r_4); // update all surfaces that this object is
        referenced to with the new rectangle pointed to by r_4
        m_surfaces[i]->delete_rectangle(this); // remove this rectangle from the associated
        surface, as it has now been updated
    }

    // check the edges, surfaces and spaces associated to this object with the new vertices:
    v_5 and v_split
    this->check_associated_members(v_5);
    this->check_associated_members(v_6);
    this->check_associated_members(v_7);
    this->check_associated_members(v_8);
    this->check_associated_members(v_split);

    m_surfaces.clear(); // clear the vector, this rectangle object now does no longer belong
    to any surface
    return;

}

Vertex_Store* Rectangle::get_store_ptr()
{
    return m_store_ptr;
}

Vector* Rectangle::get_normal_ptr()
{
    return &m_normal_vector;
}

Vertex* Rectangle::get_vertex_ptr(int n)
{
    return m_vertices[n];
}

Line* Rectangle::get_line_ptr(int n)
{
    return m_lines[n];
}

Vertex* Rectangle::get_center_vertex_ptr()
```

```
{
    return &m_center_vertex;
}

Vertex* Rectangle::surf_point_closest_vertex(Vertex* vertex_ptr)
{
    Vector vct_1(vertex_ptr, m_vertices[0]); // directional vector of the line from a point
    on the rectangle to the vertex
    Vector vct_2;

    // determine whether rectangles normal is pointing toward or away from vertex_ptr
    if (abs(m_normal_vector.get_dx()) >=
    std::max(abs(m_normal_vector.get_dy()),abs(m_normal_vector.get_dz())))
    {
        if ((vct_1.get_dx() > 0) == (m_normal_vector.get_dx() > 0))
        {
            vct_2 = m_normal_vector.change_direction();
        }
        else
        {
            vct_2 = m_normal_vector;
        }
    }
    else if (abs(m_normal_vector.get_dy()) >=
    std::max(abs(m_normal_vector.get_dx()),abs(m_normal_vector.get_dz())))
    {
        if ((vct_1.get_dy() > 0) == (m_normal_vector.get_dy() > 0))
        {
            vct_2 = m_normal_vector.change_direction();
        }
        else
        {
            vct_2 = m_normal_vector;
        }
    }
    else if (abs(m_normal_vector.get_dz()) >=
    std::max(abs(m_normal_vector.get_dz()),abs(m_normal_vector.get_dy())))
    {
        if ((vct_1.get_dz() > 0) == (m_normal_vector.get_dz() > 0))
        {
            vct_2 = m_normal_vector.change_direction();
        }
        else
        {
            vct_2 = m_normal_vector;
        }
    }
    else
    {
        std::cout << "Error in finding point on surface closest to a vertex, exiting..." <<
        std::endl;
        exit(1);
    }

    // determine the scalar for the vector of the line where the line intersects the plane
    i.e.: Intersection point = A + s.v_1
    double s = (vct_2.calc_pt_product(vct_1)) / (vct_2.calc_pt_product(vct_2));

    //compute the point with scalar s
    double x, y, z;

    x = vertex_ptr->get_x() + s * vct_2.get_dx(),
```

```
    y = vertex_ptr->get_y() + s * vct_2.get_dy(),
    z = vertex_ptr->get_z() + s * vct_2.get_dz();

    return m_store_ptr->add_vertex(x, y, z);
}

Surface* Rectangle::get_last_surface_ptr() // dangerous?
{
    return m_surfaces.back();
}

unsigned int Rectangle::get_surface_count()
{
    return m_surfaces.size();
}

Surface* Rectangle::get_surface_ptr(unsigned int n)
{
    return m_surfaces[n];
}

Edge* Rectangle::get_edge_ptr(int n)
{
    return m_lines[n]->get_last_edge_ptr();
}

Point* Rectangle::get_point_ptrs(int n)
{
    return m_vertices[n]->get_last_point_ptr();
}
```

```
Surface::Surface(Rectangle* initial_rectangle, Edge* e_1, Edge* e_2, Edge* e_3, Edge* e_4)
{
    m_rectangles.push_back(initial_rectangle);
    m_edges[0] = e_1;
    m_edges[1] = e_2;
    m_edges[2] = e_3;
    m_edges[3] = e_4;
    for (int i = 0; i < 4; i++)
    {
        m_vertices.push_back(initial_rectangle->get_vertex_ptr(i));
    }
    m_encasing_rectangle = Rectangle(&m_edges[0]->get_encasing_line(),
                                     &m_edges[1]->get_encasing_line(),
                                     &m_edges[2]->get_encasing_line(),
                                     &m_edges[3]->get_encasing_line(),
                                     nullptr);
    //ctor
}

Surface::~Surface()
{
    //dtor
}
```

```
void Surface::add_space(Space* space_ptr)
{
    m_space_ptrs.push_back(space_ptr);
    if (m_space_ptrs.size() > 1)
    {
        std::cout << "Error, surface assigned to too many spaces (only 1 possible),
        exiting.." << std::endl;
        exit(1);
    }
}

void Surface::add_rectangle(Rectangle*& rectangle_ptr)
{
    if (std::find(m_rectangles.begin(),m_rectangles.end(), rectangle_ptr) ==
    m_rectangles.end())
    {
        m_rectangles.push_back(rectangle_ptr);
    }
}

void Surface::delete_rectangle(Rectangle* rectangle_ptr)
{
    m_rectangles.erase( std::remove(m_rectangles.begin(), m_rectangles.end(),
    rectangle_ptr), m_rectangles.end());
}

unsigned int Surface::get_space_count()
{
    return m_space_ptrs.size();
}

Space* Surface::get_space_ptr(unsigned int n)
{
    return m_space_ptrs[n]; // returns the last element of this vector, which cannot be
    larger than 1 element and must be larger than 0 elements
}

Edge* Surface::get_edge_ptr(int n)
{
    return m_edges[n];
}

void Surface::check_vertex(Vertex* vertex_ptr) // checks if a vertex lies on this surface,
if so then split the rectangle on which it lies it into four new rectangles
{
    if (!m_encasing_rectangle.check_vertex(vertex_ptr)) // the vertex is not located in or
    on the cuboid
    {
        return;
    }

    if (std::find(m_vertices.begin(), m_vertices.end(), vertex_ptr) == m_vertices.end()) //
    if the vertex is not part of the surface yet, then add it to the space and continue with
    the rest of the function
    {
        m_vertices.push_back(vertex_ptr);
    }
    else // if it already is added to the surface, then no further action is required and
    the function is ended
    {
        return;
    }
```

```
    for (unsigned int i = 0; i < m_rectangles.size(); i++) // find the rectangles in which
    or on which the vertex is located
    {
        if (m_rectangles[i]->check_vertex(vertex_ptr)) // if the point is located on the
        rectangle, then split it
        {
            m_rectangles[i]->split(vertex_ptr); // split this rectangle in 2 or 4 new
            rectangles depending on where the vertex intersects
        }
    }
}

Rectangle& Surface::get_encasing_rectangle()
{
    return m_encasing_rectangle;
}
```

```
#include "../include/Vertex.h"
#include <vector>
#include <iostream>
#include <algorithm>
#include <cstdlib>

Cuboid::Cuboid(Rectangle* one, Rectangle* two, Rectangle* three, Rectangle* four, Rectangle*
five, Rectangle* six, Vertex_Store* store_ptr)
{
    m_store_ptr = store_ptr;
    m_deletion = false;

    std::fill_n(m_rectangles, 6, nullptr);
    std::fill_n(m_lines, 12, nullptr);
    std::fill_n(m_vertices, 8, nullptr);
    m_rectangles[0] = one;

    Rectangle* t_r[5];
    t_r[0] = two;
    t_r[1] = three;
    t_r[2] = four;
    t_r[3] = five;
    t_r[4] = six;

    Vector vct_1 = *m_rectangles[0]->get_normal_ptr();
    Vector vct_2(m_rectangles[0]->get_line_ptr(0)->get_vertex_ptr(0),
                 m_rectangles[0]->get_line_ptr(0)->get_vertex_ptr(1));
    Vector vct_3;
    for (int i = 1; i < 4; i++) // find a line perpendicular to vct_2
    {
        vct_3 = Vector(m_rectangles[0]->get_line_ptr(i)->get_vertex_ptr(0),
                       m_rectangles[0]->get_line_ptr(i)->get_vertex_ptr(1));
        if(vct_2.calc_pt_product(vct_3) == 0)
        {
            break;
        }
    }


    for (int i = 0; i < 5; i++)
    {
        Vector vct_4 = *t_r[i]->get_normal_ptr();
        if (vct_4.calc_cross_product(vct_1).is_zero())
        {
            m_rectangles[5] = t_r[i];
        }
        else if (vct_4.calc_pt_product(vct_2) == 0)
        {
            if (m_rectangles[1] == nullptr)
            {
                m_rectangles[1] = t_r[i];
            }
            else
            {
                m_rectangles[4] = t_r[i];
            }
        }
        else if (vct_4.calc_pt_product(vct_3) == 0)
        {
            if (m_rectangles[2] == nullptr)
            {
                m_rectangles[2] = t_r[i];
```

```
            }
            else
            {
                m_rectangles[3] = t_r[i];
            }
        }
        else
        {
            std::cout << "Error, rectangle not orthogonal during initialisation of a cuboid"
            << std::endl;
            exit(1);
        }
    }

    for (int i = 0; i < 6; i++)
    {
        if (m_rectangles[i] == nullptr)
        {
            std::cout << "Too few rectangles were assigned to a cuboid during
            initialisation, exiting.." << std::endl;
            exit(1);
        }
    }

    // add lines to m_lines
    for (int i = 0; i < 6; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            for (int k = 0; k < 12; k++)
            {
                if (m_lines[k] == nullptr)
                {
                    m_lines[k] = m_rectangles[i]->get_line_ptr(j); // it not yet in the
                    array, add it and continue with the next vertex to be added
                    break; // it is already in the array, continue with the next vertex to
                    be added
                }
                else if (*m_lines[k] == *m_rectangles[i]->get_line_ptr(j))
                {
                    break; // it is already in the array, continue with the next vertex to
                    be added
                }
                else if (k == 11)
                {
                    std::cout << "Attempted to add too many lines to a cuboid, exiting..."
                    << std::endl;
                    exit(1);
                }
            }
        }
    }

    if (m_lines[11] == nullptr)
    {
        std::cout << "Not enough lines have been added to a cuboid during initialisation,
        exiting..." << std::endl;
        exit(1); // if this occurs, then probably some lines are or are not coincident
    }

    // add vertices to m_vertices
    for (int i = 0; i < 6; i++)
```

```
    {
        for (int j = 0; j < 4; j++)
        {
            for (int k = 0; k < 8; k++)
            {
                if (m_vertices[k] == nullptr)
                {
                    m_vertices[k] = m_rectangles[i]->get_vertex_ptr(j); // it not yet in the
                    array, add it and continue with the next vertex to be added
                    break;
                }
                else if (*m_vertices[k] == *m_rectangles[i]->get_vertex_ptr(j))
                {
                    break; // it is already in the array, continue with the next vertex to
                    be added
                }
                else if (k == 7)
                {
                    std::cout << "Attempted to add too many vertices to a cuboid,
                    exiting..." << std::endl;
                    exit(1);
                }
            }
        }
    }
    if (m_vertices[7] == nullptr)
    {
        std::cout << "Not enough vertices have been added to a cuboid during initialisation,
        exiting..." << std::endl;
        exit(1); // if this occurs, then probably some vertices are or are not coincident
    }

    // calculate the center vertex
    double sum_x = 0, sum_y = 0, sum_z = 0;
    for (int i = 0; i < 8; i++)
    {
        sum_x += m_vertices[i]->get_x();
        sum_y += m_vertices[i]->get_y();
        sum_z += m_vertices[i]->get_z();
    }
    m_center_vertex = Vertex(sum_x/8.0, sum_y/8.0, sum_x/8.0);
/*
    for (int i = 0; i < 6; i++)
    {
        m_rectangles[i].m_rectangle_ptr = temp_container[i];
        Vector normal_to_center(temp_container[i]->get_center_vertex_ptr(), &m_center_vertex);
        if (temp_container[i]->get_normal_ptr()->same_direction(normal_to_center))
        {
            m_rectangles[i].m_normal_dir = true;
        }
        else
        {
            m_rectangles[i].m_normal_dir = false;
        }
    }*/
    //ctor
}

Cuboid::Cuboid()
{
    //ctor
}
```

```
Cuboid::~Cuboid()
{
    //dtor
}

bool Cuboid::operator == (const Cuboid& rhs)
{
    if (this == &rhs)
    {
        return true;
    }
    else
    {
        for (int i = 0; i < 6; i++)
        {
            if (*m_rectangles[0] == *rhs.m_rectangles[i])
            {
                for (int j = 0; j < 6; j++)
                {
                    if ((j != i) && (*m_rectangles[1] == *rhs.m_rectangles[j]))
                    {
                        for (int k = 0; k < 6; k++)
                        {
                            if ((k != j && k!= i) && (*m_rectangles[2] ==
                            *rhs.m_rectangles[k]))
                            {
                                return true; // if three surfaces of a cube are coincident
                                then the entire cube is coincident
                            }
                        }
                    }
                }
            }
        }
        return false;
    }
}

double Cuboid::get_volume()
{
    Vector vct_1(m_lines[0]->get_vertex_ptr(0), m_lines[0]->get_vertex_ptr(1));
    Vector vct_2;
    for (int i = 0; i < 6; i++)
    {
        vct_2 = *m_rectangles[i]->get_normal_ptr();
        if (vct_1.calc_cross_product(vct_2).is_zero())
        {
            return (m_rectangles[i]->get_area() * m_lines[0]->get_length());
        }
    }
    std::cout << "Error in calculating Cuboid volume, exiting..." << std::endl;
    exit(1);
}

bool Cuboid::check_vertex(Vertex* vertex_ptr)
{
    for (int i = 0; i < 8; i++)
    {
        if (*vertex_ptr == *m_vertices[i])
        {
            return false;
```

```
        }
    }

    Vector vct_1(m_vertices[0], vertex_ptr);
    Vector vct_2(m_vertices[1], vertex_ptr);
    Vector vct_3(m_vertices[2], vertex_ptr);
    Vector vct_4(m_vertices[3], vertex_ptr);
    Vector vct_5(m_vertices[4], vertex_ptr);


    int true_count_tot = 0;

    if ((vct_1.get_dx() > 0) && (vct_2.get_dx() > 0) && (vct_3.get_dx() > 0) &&
        (vct_4.get_dx() > 0) && (vct_5.get_dx() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_1.get_dx() < 0) && (vct_2.get_dx() < 0) && (vct_3.get_dx() < 0) &&
             (vct_4.get_dx() < 0) && (vct_5.get_dx() < 0))
    {
        true_count_tot++;
    }

    if ((vct_1.get_dy() > 0) && (vct_2.get_dy() > 0) && (vct_3.get_dy() > 0) &&
        (vct_4.get_dy() > 0) && (vct_5.get_dy() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_1.get_dy() < 0) && (vct_2.get_dy() < 0) && (vct_3.get_dy() < 0) &&
             (vct_4.get_dy() < 0) && (vct_5.get_dy() < 0))
    {
        true_count_tot++;
    }

    if ((vct_1.get_dz() > 0) && (vct_2.get_dz() > 0) && (vct_3.get_dz() > 0) &&
        (vct_4.get_dz() > 0) && (vct_5.get_dz() > 0))
    {
        true_count_tot++;
    }
    else if ((vct_1.get_dz() < 0) && (vct_2.get_dz() < 0) && (vct_3.get_dz() < 0) &&
             (vct_4.get_dz() < 0) && (vct_5.get_dz() < 0))
    {
        true_count_tot++;
    }

    if (true_count_tot > 0)
    {
        return false;
    }
    else
    {
        return true;
    }
}

bool Cuboid::check_deletion()
{
    return m_deletion;
}

void Cuboid::tag_deletion()
{
```

```
    m_deletion = true;
}

void Cuboid::add_space(Space* space_ptr)
{
    if (std::find(m_spaces.begin(), m_spaces.end(), space_ptr) == m_spaces.end())
    {
        m_spaces.push_back(space_ptr);
    }
}

void Cuboid::check_associated_members(Vertex* v_check)
{
    for (unsigned int i = 0; i < this->get_space_count(); i++)
    {
        for (int j = 0; j < 12; j++)
        {
            this->get_space_ptr(i)->get_edge_ptr(j)->check_vertex(v_check);
        }
        for (int j = 0; j < 6; j++)
        {
            this->get_space_ptr(i)->get_surface_ptr(j)->check_vertex(v_check);
        }
        this->get_space_ptr(i)->check_vertex(v_check);
    }
}

void Cuboid::split(Vertex* v_split)
{
    this->tag_deletion();

    for (int i = 0; i < 6; i++) // if the vertex lies on one of the cuboids lines, then the
    cuboid is to be split into two new cuboids
    { // still need to tag rectangles for deletion
        if (m_rectangles[i]->check_vertex(v_split))
        {
            Rectangle* src_rec = m_rectangles[i];

            for (int j = 0; j < 4; j++)
            {
                if (src_rec->get_line_ptr(j)->check_vertex(v_split))
                {
                    Line* src_line = m_rectangles[i]->get_line_ptr(j);

                    Vertex* v[12];
                    std::fill_n(v, 12, nullptr);
                    Line* l[20];
                    Rectangle* r[11];
                    Cuboid* c[2];

                    v[0] = src_line->get_vertex_ptr(0);
                    v[2] = v_split;
                    v[4] = src_line->get_vertex_ptr(1);

                    Vector vct_1(v[0], v[4]);
                    for (int k = 0; k < 4; k++) // find the line opposite to temp_line_ptr
                    {
                        Line* temp_ptr = src_rec->get_line_ptr(k);
                        Vector vct_2(temp_ptr->get_vertex_ptr(0),
                        temp_ptr->get_vertex_ptr(1)); // compute a vector of m_lines[k]
                        if ((j != k) && (vct_1.calc_cross_product(vct_2).is_zero())) // if a
                        line with a different index than j is parallel to that indexed with j
```

```
        {
            v[3] = temp_ptr->line_point_closest_to_vertex(v_split);
        }
        else if (j != k)
        {
            if ((*v[0] == *temp_ptr->get_vertex_ptr(0)) ||
                (*v[0] == *temp_ptr->get_vertex_ptr(1)))
            {
                if (*v[0] == *temp_ptr->get_vertex_ptr(0))
                {
                    v[1] = temp_ptr->get_vertex_ptr(1);
                }
                else
                {
                    v[1] = temp_ptr->get_vertex_ptr(0);
                }
            }
            else
            {
                if (*v[4] == *temp_ptr->get_vertex_ptr(0))
                {
                    v[5] = temp_ptr->get_vertex_ptr(1);
                }
                else
                {
                    v[5] = temp_ptr->get_vertex_ptr(0);
                }
            }
        }
    }

    if ((v[1] == nullptr) || (v[3] == nullptr) || (v[5] == nullptr))
    {
        std::cout << "Error, not all vertices were initialized during the
        splitting of a cuboid(a), exiting..." << std::endl;
        exit(1);
    }


    Vector vct_2 = *src_rec->get_normal_ptr();
    for (int k = 0; k < 6; k ++)
    {
        Vector vct_3 = *m_rectangles[k]->get_normal_ptr();
        if ((i != k) && (vct_2.calc_cross_product(vct_3).is_zero())) // if a
        plane with a different index than i is parallel to the plane with
        index i
        {
            v[6]  = m_rectangles[k]->surf_point_closest_vertex(v[0]);
            v[7]  = m_rectangles[k]->surf_point_closest_vertex(v[1]);
            v[8]  = m_rectangles[k]->surf_point_closest_vertex(v[2]);
            v[9]  = m_rectangles[k]->surf_point_closest_vertex(v[3]);
            v[10] = m_rectangles[k]->surf_point_closest_vertex(v[4]);
            v[11] = m_rectangles[k]->surf_point_closest_vertex(v[5]);
            break;
        }
        else if (k == 5)
        {
            std::cout << "Error, could not find opposite plane while
            initialising cuboid, exiting..." << std::endl;
            exit(1);
        }
    }
```

```
if ((v[6] == nullptr) || (v[7] == nullptr) || (v[8] == nullptr) || (v[9]
== nullptr) || (v[10] == nullptr) || (v[11] == nullptr))
{
    std::cout << "Error, not all vertices were initialized during the
    splitting of a cuboid(b), exiting..." << std::endl;
    exit(1);
}

for (int k = 0; k < 12; k++)
{
    Vector vct_3(m_lines[k]->get_vertex_ptr(0),
    m_lines[k]->get_vertex_ptr(1));
    if (vct_1.calc_cross_product(vct_3).is_zero())
    {
        m_lines[k]->tag_deletion(); // if the line is parallel to the
        source line, then it will be split and it needs to be tagged for
        deletion
    }
}

for (int k = 0; k < 6; k++)
{
    Vector vct_3 = *m_rectangles[k]->get_normal_ptr();
    if (vct_1.calc_pt_product(vct_3) == 0)
    {
        m_rectangles[k]->tag_deletion(); // if the rectangles normal is
        perpendicular to the source line, then it will be split and it
        needs to be tagged for deletion
    }
}

l[0]  = m_store_ptr->add_line(v[0], v[1]);
l[1]  = m_store_ptr->add_line(v[0], v[2]);
l[2]  = m_store_ptr->add_line(v[1], v[3]);
l[3]  = m_store_ptr->add_line(v[2], v[3]);
l[4]  = m_store_ptr->add_line(v[2], v[4]);
l[5]  = m_store_ptr->add_line(v[3], v[5]);
l[6]  = m_store_ptr->add_line(v[4], v[5]);

l[7]  = m_store_ptr->add_line(v[6], v[7]);
l[8]  = m_store_ptr->add_line(v[6], v[8]);
l[9]  = m_store_ptr->add_line(v[7], v[9]);
l[10] = m_store_ptr->add_line(v[8], v[9]);
l[11] = m_store_ptr->add_line(v[8], v[10]);
l[12] = m_store_ptr->add_line(v[9], v[11]);
l[13] = m_store_ptr->add_line(v[10], v[11]);

l[14] = m_store_ptr->add_line(v[0], v[6]);
l[15] = m_store_ptr->add_line(v[1], v[7]);
l[16] = m_store_ptr->add_line(v[2], v[8]);
l[17] = m_store_ptr->add_line(v[3], v[9]);
l[18] = m_store_ptr->add_line(v[4], v[10]);
l[19] = m_store_ptr->add_line(v[5], v[11]);


r[0]  = m_store_ptr->add_rectangle(l[0],  l[1],  l[2],  l[3]);
r[1]  = m_store_ptr->add_rectangle(l[3],  l[4],  l[5],  l[6]);

r[2]  = m_store_ptr->add_rectangle(l[7],  l[8],  l[9],  l[10]);
r[3]  = m_store_ptr->add_rectangle(l[10], l[11], l[12], l[13]);
```

```
                    r[4]  = m_store_ptr->add_rectangle(l[0],  l[7],  l[14], l[15]);
                    r[5]  = m_store_ptr->add_rectangle(l[1],  l[8],  l[14], l[16]);
                    r[6]  = m_store_ptr->add_rectangle(l[2],  l[9],  l[15], l[17]);
                    r[7]  = m_store_ptr->add_rectangle(l[3],  l[10], l[16], l[17]);
                    r[8]  = m_store_ptr->add_rectangle(l[4],  l[11], l[16], l[18]);
                    r[9]  = m_store_ptr->add_rectangle(l[5],  l[12], l[17], l[19]);
                    r[10] = m_store_ptr->add_rectangle(l[6],  l[13], l[18], l[19]);


                    c[0] = m_store_ptr->add_cuboid(r[0], r[2], r[4], r[5], r[6], r[7]);
                    c[1] = m_store_ptr->add_cuboid(r[1], r[3], r[7], r[8], r[9], r[10]);

                    for (unsigned int k = 0; k < m_spaces.size(); k++)
                    {
                        for (int l = 0; l < 2; l++)
                        {
                            c[l]->add_space(m_spaces[k]); // fill or update m_spaces in c[l]
                            with the space pointers contained in this object
                            m_spaces[k]->add_cuboid(c[l]); // update all spaces that this
                            object is referenced to with the new cuboid pointed to by c[l]
                        }
                        m_spaces[k]->delete_cuboid(this); // remove this cuboid from all
                        associated spaces, as these have now been updated
                    }

                    // check the edges, surfaces and spaces associated to this object with
                    the new vertices: v_09, v_10, v_11 and v_split
                    this->check_associated_members(v_split);
                    for (int k = 0; k < 12; k++)
                    {
                        this->check_associated_members(v[k]);
                    }

                    m_spaces.clear();

                    return;
                }
            }
        }
    }
    for (int i = 0; i < 6; i++) // if the vertex lies on one of the cuboids rectangles, then
    the cuboid is to be split into four new cuboids
    {
        if (m_rectangles[i]->check_vertex(v_split))
        {
            Vertex* v[17];
            std::fill_n(v, 17, nullptr);
            Line* l[33];
            Rectangle* r[20];
            Cuboid* c[4];

            Line* src_line = m_rectangles[i]->get_line_ptr(0);

            v[0] = src_line->get_vertex_ptr(0);
            v[1] = src_line->get_vertex_ptr(1);
            v[9] = src_line->line_point_closest_to_vertex(v_split);

            Vector vct_1(v[0], v[1]);
            for (int j = 1; j < 4; j++)
            {
                Line* temp_ptr = m_rectangles[i]->get_line_ptr(j);
                Vector vct_2(temp_ptr->get_vertex_ptr(0), temp_ptr->get_vertex_ptr(1));
```

```
        if (vct_1.calc_cross_product(vct_2).is_zero()) // if line j is parallel to
        line i
        {
            v[10] = temp_ptr->line_point_closest_to_vertex(v_split);
        }
        else // else it is perpendicular
        {
            if ((*v[0] == *temp_ptr->get_vertex_ptr(0)) || (*v[0] ==
            *temp_ptr->get_vertex_ptr(1)))
            {
                v[11] = temp_ptr->line_point_closest_to_vertex(v_split);
                if (*v[0] == *temp_ptr->get_vertex_ptr(0))
                {
                    v[2] = temp_ptr->get_vertex_ptr(1);
                }
                else
                {
                    v[2] = temp_ptr->get_vertex_ptr(0);
                }
            }
            else
            {
                v[12] = temp_ptr->line_point_closest_to_vertex(v_split);
                if (*v[1] == *temp_ptr->get_vertex_ptr(0))
                {
                    v[3] = temp_ptr->get_vertex_ptr(1);
                }
                else
                {
                    v[3] = temp_ptr->get_vertex_ptr(0);
                }
            }
        }
    }

    if ((v[0] == nullptr) || (v[1] == nullptr) || (v[2] == nullptr) || (v[3] ==
    nullptr) ||
        (v[9] == nullptr) || (v[10] == nullptr) || (v[11] == nullptr)|| (v[12] ==
        nullptr))
    {
        std::cout << "Error, not all vertices were initialized during the splitting
        of a cuboid(c), exiting..." << std::endl;
        exit(1);
    }

    Vector vct_2 = *m_rectangles[i]->get_normal_ptr();
    for (int j = 0; j < 6; j ++)
    {
        Vector vct_3 = *m_rectangles[j]->get_normal_ptr();
        if ((i != j) && (vct_2.calc_cross_product(vct_3).is_zero())) // if a plane
        with a different index than i is parallel to the plane with index i
        {
            v[4]  = m_rectangles[j]->surf_point_closest_vertex(v[0]);
            v[5]  = m_rectangles[j]->surf_point_closest_vertex(v[2]);
            v[6]  = m_rectangles[j]->surf_point_closest_vertex(v[3]);
            v[7]  = m_rectangles[j]->surf_point_closest_vertex(v[1]);
            v[8]  = m_rectangles[j]->surf_point_closest_vertex(v_split);
            v[13] = m_rectangles[j]->surf_point_closest_vertex(v[9]);
            v[14] = m_rectangles[j]->surf_point_closest_vertex(v[10]);
            v[15] = m_rectangles[j]->surf_point_closest_vertex(v[11]);
            v[16] = m_rectangles[j]->surf_point_closest_vertex(v[12]);
        }
```

```
        }

        if ((v[4] == nullptr) || (v[5] == nullptr) || (v[6] == nullptr) || (v[7] ==
        nullptr) ||
            (v[8] == nullptr) || (v[13] == nullptr) || (v[14] == nullptr)|| (v[15] ==
            nullptr)|| (v[16] == nullptr))
        {
            std::cout << "Error, not all vertices were initialized during the splitting
            of a cuboid(d), exiting..." << std::endl;
            exit(1);
        }

        for (int j = 0; j < 12; j++)
        {
            Vector vct_3(m_lines[j]->get_vertex_ptr(0), m_lines[j]->get_vertex_ptr(1));
            if (vct_2.calc_pt_product(vct_3) == 0)
            {
                m_lines[j]->tag_deletion(); // if the line is perpendicular to the
                normal of the rectangle on which v_split lies, then it will be split and
                it needs to be tagged for deletion
            }
        }

        for (int j = 0; j < 6; j++)
        {
            m_rectangles[j]->tag_deletion();
        }

        l[0]  = m_store_ptr->add_line(v[0],  v[4]);
        l[1]  = m_store_ptr->add_line(v[11], v[15]);
        l[2]  = m_store_ptr->add_line(v[2],  v[5]);
        l[3]  = m_store_ptr->add_line(v[9] , v[13]);
        l[4]  = m_store_ptr->add_line(v[8],  v_split);
        l[5]  = m_store_ptr->add_line(v[10], v[14]);
        l[6]  = m_store_ptr->add_line(v[1],  v[7]);
        l[7]  = m_store_ptr->add_line(v[12], v[16]);
        l[8]  = m_store_ptr->add_line(v[3],  v[6]);

        l[9]  = m_store_ptr->add_line(v[0],  v[11]);
        l[10] = m_store_ptr->add_line(v[11], v[2]);
        l[11] = m_store_ptr->add_line(v[0],  v[9]);
        l[12] = m_store_ptr->add_line(v[11], v_split);
        l[13] = m_store_ptr->add_line(v[2],  v[10]);
        l[14] = m_store_ptr->add_line(v[9] , v_split);
        l[15] = m_store_ptr->add_line(v[10], v_split);
        l[16] = m_store_ptr->add_line(v[9] , v[1]);
        l[17] = m_store_ptr->add_line(v[12], v_split);
        l[18] = m_store_ptr->add_line(v[10], v[3]);
        l[19] = m_store_ptr->add_line(v[1],  v[12]);
        l[20] = m_store_ptr->add_line(v[12], v[3]);

        l[21] = m_store_ptr->add_line(v[4],  v[15]);
        l[22] = m_store_ptr->add_line(v[15], v[5]);
        l[23] = m_store_ptr->add_line(v[4],  v[13]);
        l[24] = m_store_ptr->add_line(v[15], v[8]);
        l[25] = m_store_ptr->add_line(v[5],  v[14]);
        l[26] = m_store_ptr->add_line(v[13], v[8]);
        l[27] = m_store_ptr->add_line(v[8],  v[14]);
        l[28] = m_store_ptr->add_line(v[13], v[7]);
        l[29] = m_store_ptr->add_line(v[8],  v[16]);
        l[30] = m_store_ptr->add_line(v[14], v[6]);
        l[31] = m_store_ptr->add_line(v[7],  v[16]);
```

```
        l[32] = m_store_ptr->add_line(v[16], v[6]);


        r[0]  = m_store_ptr->add_rectangle(l[0],  l[1],  l[9] , l[21]);
        r[1]  = m_store_ptr->add_rectangle(l[1],  l[2],  l[10], l[22]);
        r[2]  = m_store_ptr->add_rectangle(l[0],  l[3],  l[11], l[23]);
        r[3]  = m_store_ptr->add_rectangle(l[1],  l[4],  l[12], l[24]);
        r[4]  = m_store_ptr->add_rectangle(l[2],  l[5],  l[13], l[25]);
        r[5]  = m_store_ptr->add_rectangle(l[3],  l[4],  l[14], l[26]);
        r[6]  = m_store_ptr->add_rectangle(l[4],  l[5],  l[15], l[27]);
        r[7]  = m_store_ptr->add_rectangle(l[3],  l[6],  l[16], l[28]);
        r[8]  = m_store_ptr->add_rectangle(l[4],  l[7],  l[17], l[29]);
        r[9]  = m_store_ptr->add_rectangle(l[5],  l[8],  l[18], l[30]);
        r[10] = m_store_ptr->add_rectangle(l[6],  l[7],  l[19], l[31]);
        r[11] = m_store_ptr->add_rectangle(l[7],  l[8],  l[20], l[32]);

        r[12] = m_store_ptr->add_rectangle(l[9] , l[11], l[12], l[14]);
        r[13] = m_store_ptr->add_rectangle(l[10], l[12], l[13], l[15]);
        r[14] = m_store_ptr->add_rectangle(l[14], l[16], l[17], l[19]);
        r[15] = m_store_ptr->add_rectangle(l[15], l[17], l[18], l[20]);

        r[16] = m_store_ptr->add_rectangle(l[21], l[23], l[24], l[26]);
        r[17] = m_store_ptr->add_rectangle(l[22], l[24], l[25], l[27]);
        r[18] = m_store_ptr->add_rectangle(l[26], l[28], l[29], l[31]);
        r[19] = m_store_ptr->add_rectangle(l[27], l[29], l[30], l[32]);


        c[0] = m_store_ptr->add_cuboid(r[0], r[2], r[3],  r[5],  r[12], r[16]);
        c[1] = m_store_ptr->add_cuboid(r[1], r[3], r[4],  r[6],  r[13], r[17]);
        c[2] = m_store_ptr->add_cuboid(r[5], r[7], r[8],  r[10], r[14], r[18]);
        c[3] = m_store_ptr->add_cuboid(r[6], r[8], r[9],  r[11], r[15], r[19]);

        for (unsigned int j = 0; j < m_spaces.size(); j++)
        {
            for (int k = 0; k < 4; k++)
            {
                c[k]->add_space(m_spaces[j]); // fill or update m_spaces in c[k] with
                the space pointers contained in this object
                m_spaces[j]->add_cuboid(c[k]); // update all spaces that this object is
                referenced to with the new cuboid pointed to by c[k]
            }
            m_spaces[j]->delete_cuboid(this); // remove this cuboid from all associated
            spaces, as these have now been updated
        }

        // check the edges, surfaces and spaces associated to this object with the new
        vertices: v_09, v_10, v_11 and v_split

        this->check_associated_members(v_split);
        for (int j = 0; j < 17; j++)
        {
            this->check_associated_members(v[j]);
        }

        m_spaces.clear();

        return;
    }
}// if none of the above applies, then do one final check to see if the vertex is really
in the cuboid

Vertex* v[26];
```

```
    std::fill_n(v, 26, nullptr);
    Line* l[53];
    Rectangle* r[35];
    Cuboid* c[8];

    Rectangle* src_rec = m_rectangles[0];
    Line* src_line = src_rec->get_line_ptr(0);

    v[4] = src_rec->surf_point_closest_vertex(v_split);
    v[0] = src_line->get_vertex_ptr(0);
    v[3] = src_line->line_point_closest_to_vertex(v[4]);
    v[6] = src_line->get_vertex_ptr(1);

    Vector vct_1(v[0], v[6]);
    for (int i = 1; i < 4; i++)
    {
        Line* temp_ptr = src_rec->get_line_ptr(i);
        Vector vct_2(temp_ptr->get_vertex_ptr(0), temp_ptr->get_vertex_ptr(1));
        if (vct_1.calc_cross_product(vct_2).is_zero()) // if the lines are parallel
        {
            v[5] = m_lines[i]->line_point_closest_to_vertex(v[4]);
        }
        else
        {
            if ((*v[0] == *temp_ptr->get_vertex_ptr(0)) || (*v[0] ==
            *temp_ptr->get_vertex_ptr(1)))
            {
                v[1] = temp_ptr->line_point_closest_to_vertex(v[4]);
                if (*v[0] == *temp_ptr->get_vertex_ptr(0))
                {
                    v[2] = temp_ptr->get_vertex_ptr(1);
                }
                else
                {
                    v[2] = temp_ptr->get_vertex_ptr(0);
                }
            }
            else
            {
                v[7] = temp_ptr->line_point_closest_to_vertex(v[4]);
                if (*v[6] == *temp_ptr->get_vertex_ptr(0))
                {
                    v[8] = temp_ptr->get_vertex_ptr(1);
                }
                else
                {
                    v[8] = temp_ptr->get_vertex_ptr(0);
                }
            }
        }
    }

    if ((v[0] == nullptr) || (v[1] == nullptr) || (v[2] == nullptr) || (v[3] == nullptr) ||
        (v[4] == nullptr) || (v[5] == nullptr) || (v[6] == nullptr)|| (v[7] == nullptr) ||
        (v[8] == nullptr))
    {
        std::cout << "Error, not all vertices were initialized during the splitting of a
        cuboid(e), exiting..." << std::endl;
        exit(1);
    }

    Vector vct_2 = *src_rec->get_normal_ptr();
```

```
    for (int i = 1; i < 6; i ++)
    {
        Vector vct_3 = *m_rectangles[i]->get_normal_ptr();
        if (vct_2.calc_cross_product(vct_3).is_zero()) // if the plane is parallel to the
        source plane
        {
            v[17] = m_rectangles[i]->surf_point_closest_vertex(v[0]);
            v[18] = m_rectangles[i]->surf_point_closest_vertex(v[1]);
            v[19] = m_rectangles[i]->surf_point_closest_vertex(v[2]);
            v[20] = m_rectangles[i]->surf_point_closest_vertex(v[3]);
            v[21] = m_rectangles[i]->surf_point_closest_vertex(v[4]);
            v[22] = m_rectangles[i]->surf_point_closest_vertex(v[5]);
            v[23] = m_rectangles[i]->surf_point_closest_vertex(v[6]);
            v[24] = m_rectangles[i]->surf_point_closest_vertex(v[7]);
            v[25] = m_rectangles[i]->surf_point_closest_vertex(v[8]);
        }
    }

    if ((v[17] == nullptr) || (v[18] == nullptr) || (v[19] == nullptr) || (v[20] == nullptr)
    ||
        (v[21] == nullptr) || (v[22] == nullptr) || (v[23] == nullptr) || (v[24] == nullptr)
        || (v[25] == nullptr))
    {
        std::cout << "Error, not all vertices were initialized during the splitting of a
        cuboid(e), exiting..." << std::endl;
        exit(1);
    }

    for (int i = 0; i < 12; i++)
    {
        m_lines[i]->tag_deletion();
    }

    for (int i = 0; i < 6; i++)
    {
        m_rectangles[i]->tag_deletion();
    }

    v[9]  = (m_store_ptr->add_line(v[0] , v[17]))->line_point_closest_to_vertex(v_split);
    v[10] = (m_store_ptr->add_line(v[1] , v[18]))->line_point_closest_to_vertex(v_split);
    v[11] = (m_store_ptr->add_line(v[2] , v[19]))->line_point_closest_to_vertex(v_split);
    v[12] = (m_store_ptr->add_line(v[3] , v[20]))->line_point_closest_to_vertex(v_split);

    v[13] = (m_store_ptr->add_line(v[5] , v[22]))->line_point_closest_to_vertex(v_split);
    v[14] = (m_store_ptr->add_line(v[6] , v[23]))->line_point_closest_to_vertex(v_split);
    v[15] = (m_store_ptr->add_line(v[7] , v[24]))->line_point_closest_to_vertex(v_split);
    v[16] = (m_store_ptr->add_line(v[8] , v[25]))->line_point_closest_to_vertex(v_split);


    l[0]  = m_store_ptr->add_line(v[0], v[1]);
    l[1]  = m_store_ptr->add_line(v[1], v[2]);
    l[2]  = m_store_ptr->add_line(v[0], v[3]);
    l[3]  = m_store_ptr->add_line(v[1], v[4]);
    l[4]  = m_store_ptr->add_line(v[2], v[5]);
    l[5]  = m_store_ptr->add_line(v[3], v[4]);
    l[6]  = m_store_ptr->add_line(v[4], v[5]);
    l[7]  = m_store_ptr->add_line(v[3], v[6]);
    l[8]  = m_store_ptr->add_line(v[4], v[7]);
    l[9]  = m_store_ptr->add_line(v[5], v[8]);
    l[10] = m_store_ptr->add_line(v[6], v[7]);
    l[11] = m_store_ptr->add_line(v[7], v[8]);
```

```
    l[12] = m_store_ptr->add_line(v[9], v[10]);
    l[13] = m_store_ptr->add_line(v[10], v[11]);
    l[14] = m_store_ptr->add_line(v[9], v[12]);
    l[15] = m_store_ptr->add_line(v[10], v_split);
    l[16] = m_store_ptr->add_line(v[11], v[13]);
    l[17] = m_store_ptr->add_line(v[12], v_split);
    l[18] = m_store_ptr->add_line(v_split, v[13]);
    l[19] = m_store_ptr->add_line(v[12], v[14]);
    l[20] = m_store_ptr->add_line(v_split, v[15]);
    l[21] = m_store_ptr->add_line(v[13], v[16]);
    l[22] = m_store_ptr->add_line(v[14], v[15]);
    l[23] = m_store_ptr->add_line(v[15], v[16]);

    l[24] = m_store_ptr->add_line(v[17], v[18]);
    l[25] = m_store_ptr->add_line(v[18], v[19]);
    l[26] = m_store_ptr->add_line(v[17], v[20]);
    l[27] = m_store_ptr->add_line(v[18], v[21]);
    l[28] = m_store_ptr->add_line(v[19], v[22]);
    l[29] = m_store_ptr->add_line(v[20], v[21]);
    l[30] = m_store_ptr->add_line(v[21], v[22]);
    l[31] = m_store_ptr->add_line(v[20], v[23]);
    l[32] = m_store_ptr->add_line(v[21], v[24]);
    l[33] = m_store_ptr->add_line(v[22], v[25]);
    l[34] = m_store_ptr->add_line(v[23], v[24]);
    l[35] = m_store_ptr->add_line(v[24], v[25]);

    l[36] = m_store_ptr->add_line(v[0], v[9]);
    l[37] = m_store_ptr->add_line(v[1], v[10]);
    l[38] = m_store_ptr->add_line(v[2], v[11]);
    l[39] = m_store_ptr->add_line(v[3], v[12]);
    l[40] = m_store_ptr->add_line(v[4], v_split);
    l[41] = m_store_ptr->add_line(v[5], v[13]);
    l[42] = m_store_ptr->add_line(v[6], v[14]);
    l[43] = m_store_ptr->add_line(v[7], v[15]);
    l[44] = m_store_ptr->add_line(v[8], v[16]);

    l[45] = m_store_ptr->add_line(v[9], v[17]);
    l[46] = m_store_ptr->add_line(v[10], v[18]);
    l[47] = m_store_ptr->add_line(v[11], v[19]);
    l[48] = m_store_ptr->add_line(v[12], v[20]);
    l[49] = m_store_ptr->add_line(v_split, v[21]);
    l[50] = m_store_ptr->add_line(v[13], v[22]);
    l[51] = m_store_ptr->add_line(v[14], v[23]);
    l[52] = m_store_ptr->add_line(v[15], v[24]);
    l[53] = m_store_ptr->add_line(v[16], v[25]);


    r[0]  = m_store_ptr->add_rectangle(l[0] , l[2] , l[3] , l[5] );
    r[1]  = m_store_ptr->add_rectangle(l[1] , l[3] , l[4] , l[6] );
    r[2]  = m_store_ptr->add_rectangle(l[5] , l[7] , l[8] , l[10]);
    r[3]  = m_store_ptr->add_rectangle(l[6] , l[8] , l[9] , l[11]);

    r[4]  = m_store_ptr->add_rectangle(l[12], l[14], l[15], l[17]);
    r[5]  = m_store_ptr->add_rectangle(l[13], l[15], l[16], l[18]);
    r[6]  = m_store_ptr->add_rectangle(l[17], l[19], l[20], l[22]);
    r[7]  = m_store_ptr->add_rectangle(l[18], l[20], l[21], l[23]);

    r[8]  = m_store_ptr->add_rectangle(l[24], l[26], l[27], l[29]);
    r[9]  = m_store_ptr->add_rectangle(l[25], l[27], l[28], l[30]);
    r[10] = m_store_ptr->add_rectangle(l[29], l[31], l[32], l[34]);
    r[11] = m_store_ptr->add_rectangle(l[30], l[32], l[33], l[35]);
```

```
    r[12] = m_store_ptr->add_rectangle(l[36], l[37], l[0] , l[12]);
    r[13] = m_store_ptr->add_rectangle(l[37], l[38], l[1] , l[13]);
    r[14] = m_store_ptr->add_rectangle(l[36], l[39], l[2] , l[14]);
    r[15] = m_store_ptr->add_rectangle(l[37], l[40], l[3] , l[15]);
    r[16] = m_store_ptr->add_rectangle(l[38], l[41], l[4] , l[16]);
    r[17] = m_store_ptr->add_rectangle(l[39], l[40], l[5] , l[17]);
    r[18] = m_store_ptr->add_rectangle(l[40], l[41], l[6] , l[18]);
    r[19] = m_store_ptr->add_rectangle(l[39], l[42], l[7] , l[19]);
    r[20] = m_store_ptr->add_rectangle(l[40], l[43], l[8] , l[20]);
    r[21] = m_store_ptr->add_rectangle(l[41], l[44], l[9] , l[21]);
    r[22] = m_store_ptr->add_rectangle(l[42], l[43], l[10], l[22]);
    r[23] = m_store_ptr->add_rectangle(l[43], l[44], l[11], l[23]);

    r[24] = m_store_ptr->add_rectangle(l[45], l[46], l[12], l[24]);
    r[25] = m_store_ptr->add_rectangle(l[46], l[47], l[13], l[25]);
    r[26] = m_store_ptr->add_rectangle(l[45], l[48], l[14], l[26]);
    r[27] = m_store_ptr->add_rectangle(l[46], l[49], l[15], l[27]);
    r[28] = m_store_ptr->add_rectangle(l[47], l[50], l[16], l[28]);
    r[29] = m_store_ptr->add_rectangle(l[48], l[49], l[17], l[29]);
    r[30] = m_store_ptr->add_rectangle(l[49], l[50], l[18], l[30]);
    r[31] = m_store_ptr->add_rectangle(l[48], l[51], l[19], l[31]);
    r[32] = m_store_ptr->add_rectangle(l[49], l[52], l[20], l[32]);
    r[33] = m_store_ptr->add_rectangle(l[50], l[53], l[21], l[33]);
    r[34] = m_store_ptr->add_rectangle(l[51], l[52], l[22], l[34]);
    r[35] = m_store_ptr->add_rectangle(l[52], l[53], l[23], l[35]);


    c[0] = m_store_ptr->add_cuboid(r[0] , r[4] , r[12], r[14], r[15], r[17]);
    c[1] = m_store_ptr->add_cuboid(r[1] , r[5] , r[13], r[15], r[16], r[18]);
    c[2] = m_store_ptr->add_cuboid(r[2] , r[6] , r[17], r[19], r[20], r[22]);
    c[3] = m_store_ptr->add_cuboid(r[3] , r[7] , r[18], r[20], r[21], r[23]);

    c[4] = m_store_ptr->add_cuboid(r[4] , r[8] , r[24], r[26], r[27], r[29]);
    c[5] = m_store_ptr->add_cuboid(r[5] , r[9] , r[25], r[27], r[28], r[30]);
    c[6] = m_store_ptr->add_cuboid(r[6] , r[10], r[29], r[31], r[32], r[34]);
    c[7] = m_store_ptr->add_cuboid(r[7] , r[11], r[30], r[32], r[33], r[35]);


    for (unsigned int i = 0; i < m_spaces.size(); i++)
    {
        for (int j = 0 ; j < 8; j++)
        {
            c[j]->add_space(m_spaces[i]); // fill or update m_spaces in c[j] with the space
            pointers contained in this object
            m_spaces[i]->add_cuboid(c[j]); // update all spaces that this object is
            referenced to with the new cuboid pointed to by c[j]
        }
        m_spaces[i]->delete_cuboid(this); // remove this cuboid from all associated spaces,
        as these have now been updated
    }

    // check the edges, surfaces and spaces associated to this object with the new vertices:
    v_09, v_10, v_11 and v_split
    this->check_associated_members(v_split);
    for (int i = 0; i < 26; i++) // index 1-7 are the original vertices
    {
        this->check_associated_members(v[i]);
    }

    m_spaces.clear();

    return;
```

```
}

Vertex_Store* Cuboid::get_store_ptr()
{
    return m_store_ptr;
}

unsigned int Cuboid::get_space_count()
{
    return m_spaces.size();
}

Vertex* Cuboid::get_center_vertex_ptr()
{
    return &m_center_vertex;
}

Vertex* Cuboid::get_vertex_ptr(int n)
{
    return m_vertices[n];
}

Vertex* Cuboid::get_max_vertex()
{
    Vertex* temp_ptr = m_vertices[0];
    for (int i = 1; i <8; i++)
    {
        if ((temp_ptr->get_x() < m_vertices[i]->get_x()) ||
            (temp_ptr->get_y() < m_vertices[i]->get_y()) ||
            (temp_ptr->get_z() < m_vertices[i]->get_z()))
        {
            temp_ptr = m_vertices[i];
        }
    }
    return temp_ptr;
}

Vertex* Cuboid::get_min_vertex()
{
    Vertex* temp_ptr = m_vertices[0];
    for (int i = 1; i < 8; i++)
    {
        if ((temp_ptr->get_x() > m_vertices[i]->get_x()) ||
            (temp_ptr->get_y() > m_vertices[i]->get_y()) ||
            (temp_ptr->get_z() > m_vertices[i]->get_z()))
        {
            temp_ptr = m_vertices[i];
        }
    }
    return temp_ptr;
}

Line* Cuboid::get_line_ptr(int n)
{
    return m_lines[n];
}

Rectangle* Cuboid::get_rectangle_ptr(int n)
{
    return m_rectangles[n];
}
```

```
Space* Cuboid::get_last_space_ptr()
{
    return m_spaces.back();
}

Space* Cuboid::get_space_ptr(unsigned int n)
{
    return m_spaces[n];
}

Surface* Cuboid::get_surface_ptrs(int n)
{
    return m_rectangles[n]->get_last_surface_ptr();
}

Edge* Cuboid::get_edge_ptrs(int n)
{
    return m_lines[n]->get_last_edge_ptr();
}

Point* Cuboid::get_point_ptrs(int n)
{
    return m_vertices[n]->get_last_point_ptr();
}




Space::Space(int room_ID, Cuboid* initial_cuboid, Surface* s_1, Surface* s_2, Surface* s_3,
Surface* s_4, Surface* s_5, Surface* s_6)
{
    m_space_ID = room_ID;
    m_cuboids.push_back(initial_cuboid);
    std::fill_n(m_edges, 12, nullptr);
    std::fill_n(m_points, 8, nullptr);

    m_surfaces[0] = s_1;
    m_surfaces[1] = s_2;
    m_surfaces[2] = s_3;
    m_surfaces[3] = s_4;
    m_surfaces[4] = s_5;
    m_surfaces[5] = s_6;

    for (int i = 0; i < 8; i++)
    {
        m_vertices.push_back(initial_cuboid->get_vertex_ptr(i));
    }

    m_encasing_cuboid = Cuboid(&m_surfaces[0]->get_encasing_rectangle(),
                               &m_surfaces[1]->get_encasing_rectangle(),
                               &m_surfaces[2]->get_encasing_rectangle(),
                               &m_surfaces[3]->get_encasing_rectangle(),
                               &m_surfaces[4]->get_encasing_rectangle(),
                               &m_surfaces[5]->get_encasing_rectangle(),
                              nullptr);

    for (int i = 0; i < 6; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            for (int k = 0; k < 12; k++)
```

```
            {
                if (m_surfaces[i]->get_edge_ptr(j) == m_edges[k])
                {
                    break; // it is already in the array, break
                }
                else if (m_edges[k] == nullptr)
                {
                    m_edges[k] = m_surfaces[i]->get_edge_ptr(j);
                }
                else if (k == 12)
                {
                    std::cout << "Too many edges encountered while initializing a space,
                    exiting..." << std::endl;
                    exit(1);
                }
            }
        }
        /*for (int j = 0; j < 4; j++)
        {
            for (int k = 0; k < 8; k++)
            {
                if (m_surfaces[i]->get_point_ptr(j) == m_points[k])
                {
                    break; // it is already in the array, break
                }
                else if (m_edges[k] == nullptr)
                {
                    m_points[k] = m_surfaces[i]->get_point_ptr(j);
                }
                else if (k == 8)
                {
                    std::cout << "Too many points encountered while initializing a space,
                    exiting..." << std::endl;
                    exit(1);
                }
            }
        }*/
    }

    if (m_edges[11] == nullptr)
    {
        std::cout << "Too few edges found while initializing a space, exiting..." <<
        std::endl;
        exit(1);
    }

    /*if (m_points[7] == nullptr)
    {
        std::cout << "Too few points found while initializing a space, exiting..." <<
        std::endl;
        exit(1);
    }*/
    //ctor
}

Space::~Space()
{
    //dtor
}

int Space::get_ID()
{
```

```
    return m_space_ID;
}

void Space::add_cuboid(Cuboid* cuboid_ptr)
{
    if (std::find(m_cuboids.begin(),m_cuboids.end(), cuboid_ptr) == m_cuboids.end())
    {
        m_cuboids.push_back(cuboid_ptr);
    }
}

void Space::delete_cuboid(Cuboid* cuboid_ptr)
{
    m_cuboids.erase( std::remove(m_cuboids.begin(), m_cuboids.end(), cuboid_ptr),
    m_cuboids.end());
}

Edge* Space::get_edge_ptr(int n)
{
    return m_edges[n];
}

Surface* Space::get_surface_ptr(int n)
{
    return m_surfaces[n];
}

void Space::check_vertex(Vertex* vertex_ptr)
{
    if (!m_encasing_cuboid.check_vertex(vertex_ptr)) // the vertex is not located in or on
    the cuboid
    {
        return;
    }

    if (std::find(m_vertices.begin(), m_vertices.end(), vertex_ptr) == m_vertices.end()) //
    if the vertex is not part of the space yet, then add it to the space
    {
        m_vertices.push_back(vertex_ptr);
    }
    else // if it already is added to the space, then no further action is required and the
    function is ended
    {
        return;
    }

    for (unsigned int i = 0; i < m_cuboids.size(); i++) // find the cubes in which or on
    which the vertex is located
    {
        if (m_cuboids[i]->check_vertex(vertex_ptr))
        {
           m_cuboids[i]->split(vertex_ptr); // split this cuboid in 2,4 or 8 new rectangles
           depending on where the vertex intersects
        }
    }
}

Cuboid& Space::get_encasing_cuboid()
{
    return m_encasing_cuboid;
}
```

```cpp
#include "../include/Vertex.h"
#include <iostream>
#include <algorithm>


Vertex_Store::Vertex_Store()
{
    //ctor
}

Vertex_Store::~Vertex_Store()
{
    for (unsigned int i = 0; i < m_vertices.size(); i++)
    {
        delete m_vertices[i];
        m_vertices.clear();
    }

    for (unsigned int i = 0; i < m_lines.size(); i++)
    {
        delete m_lines[i];
        m_lines.clear();
    }

    for (unsigned int i = 0; i < m_rectangles.size(); i++)
    {
        delete m_rectangles[i];
        m_rectangles.clear();
    }

    for (unsigned int i = 0; i < m_cubes.size(); i++)
    {
        delete m_cubes[i];
        m_rectangles.clear();
    }
    //dtor
}

Vertex* Vertex_Store::add_vertex(double x, double y, double z)
{
    Vertex* temp_ptr = new Vertex(x, y, z);
    bool ptr_found = false;
    for (unsigned int i = 0; i < m_vertices.size(); i++)
    {
        if (*temp_ptr == *(m_vertices[i]))
        {
            ptr_found = true;

            delete temp_ptr;
            return m_vertices[i];
        }
    }
    if (!ptr_found)
    {
        m_vertices.push_back(temp_ptr);
        return temp_ptr;
    }
    else
    {
        std::cout << "Error in initializing vertex, exiting..." << std::endl;
        exit(1);
    }
```

```
}

Line* Vertex_Store::add_line(Vertex* one, Vertex* two)
{
    Line* temp_ptr = new Line(one, two, this);
    bool ptr_found = false;
    for (unsigned int i = 0; i < m_lines.size(); i++)
    {
        if (*temp_ptr == *(m_lines[i]))
        {
            ptr_found = true;

            delete temp_ptr;
            return m_lines[i];
        }
    }
    if (!ptr_found)
    {
        m_lines.push_back(temp_ptr);
        return temp_ptr;
    }
    else
    {
        std::cout << "Error in initializing line, exiting..." << std::endl;
        exit(1);
    }
}

Rectangle* Vertex_Store::add_rectangle(Line* one, Line* two, Line* three, Line* four)
{
    Rectangle* temp_ptr = new Rectangle(one, two, three, four, this);
    bool ptr_found = false;
    for (unsigned int i = 0; i < m_rectangles.size(); i++)
    {
        if (*temp_ptr == *(m_rectangles[i]))
        {
            ptr_found = true;

            delete temp_ptr;
            return m_rectangles[i];
        }
    }
    if (!ptr_found)
    {
        m_rectangles.push_back(temp_ptr);
        return temp_ptr;
    }
    else
    {
        std::cout << "Error in initializing rectangle, exiting..." << std::endl;
        exit(1);
    }
}

Cuboid* Vertex_Store::add_cuboid(Rectangle* one, Rectangle* two, Rectangle* three,
Rectangle* four, Rectangle* five, Rectangle* six)
{
    Cuboid* temp_ptr = new Cuboid(one, two, three, four, five, six, this);
    bool ptr_found = false;
    for (unsigned int i = 0; i < m_cubes.size(); i++)
    {
        if (*temp_ptr == *(m_cubes[i]))
```

```
        {
            ptr_found = true;

            delete temp_ptr;
            return m_cubes[i];
        }
    }
    if (!ptr_found)
    {
        m_cubes.push_back(temp_ptr);
        return temp_ptr;
    }
    else
    {
        std::cout << "Error in initializing cuboid, exiting..." << std::endl;
        exit(1);
    }
}

void Vertex_Store::delete_line(Line* l_d)
{
    m_lines.erase(std::remove(m_lines.begin(), m_lines.end(), l_d), m_lines.end()); //
    remove the element of the vector containing the address of this pointer
    delete l_d; // release the memory at the address pointed to by this pointer
}

void Vertex_Store::delete_rectangle(Rectangle* r_d)
{
    m_rectangles.erase( std::remove(m_rectangles.begin(), m_rectangles.end(), r_d),
    m_rectangles.end()); // remove the element of the vector containing the address of this
    pointer
    delete r_d; // release the memory at the address pointed to by this pointer
}

void Vertex_Store::delete_cuboid(Cuboid* c_d)
{
    m_cubes.erase(std::remove(m_cubes.begin(), m_cubes.end(), c_d), m_cubes.end()); //
    remove the element of the vector containing the address of this pointer
    delete c_d; // release the memory at the address pointed to by this pointer
}
```

```
#ifndef BUILDING_H
#define BUILDING_H

#include <Vertex.h>
#include <string>

struct BP_space;
struct BP_wall;
struct BP_window;
struct BP_floor;

class Building : public Vertex_Store
{
private:
    std::vector<Point*> m_points;
    std::vector<Edge*> m_edges;
    std::vector<Surface*> m_surfaces;
    std::vector<Space*> m_spaces;

    std::vector<BP_space> m_BP_spaces;
    std::vector<BP_wall> m_BP_walls;
    std::vector<BP_window> m_BP_windows;
    std::vector<BP_floor> m_BP_floors;
public:
    Building(std::string file_name);
    ~Building();

    void make_conformal();
    void make_BP_input();
    void write_BP_input(std::string file_name);
    void write_blocks(std::string file_name);

    Point* add_point(Vertex* vertex_ptr);
    void add_point(double x, double y, double z);
    Edge* add_edge(Line* line_ptr, Point* p_1, Point* p_2);
    void add_edge(double x1, double y1, double z1, double x2, double y2, double z2); //
    orthogonal only
    Surface* add_surface(Rectangle* rectangle_ptr, Edge* e_1, Edge* e_2, Edge* e_3, Edge*
    e_4);
    void add_surface(double x1, double y1, double z1, double x2, double y2, double z2); //
    orthogonal only
    void add_space(int room_ID, double x, double y, double z, double w, double d, double h);
    // orthogonal only

    Surface* get_surface(int n);
    Space* get_space(int n);

    Cuboid* get_cuboid(int n);
    Rectangle* get_rectangle(int n);
    Line* get_line(int n);
};

struct BP_wall
{
    int m_ID;
    int m_construction_ID;
    double m_area;
    double m_orientation;
    std::string m_space_side_1, m_space_side_2;
};
struct BP_window
{
```

```
    int m_ID;
    double m_area;
    double m_U_value;
    double m_capacitance_per_area;
    double m_orientation;
    std::string m_space_side_1, m_space_side_2;
};
struct BP_floor
{
    int m_ID;
    int m_construction_ID;
    double m_area;
    std::string m_space_side_1, m_space_side_2;
};
struct BP_space
{
    int m_ID;
    double m_volume;
    double m_Q_heat, m_Q_cool;
    double m_T_heat, m_T_cool;
    double m_ACH;
};

#endif // BUILDING_H
```

```cpp
#include "../include/Building.h"
#include "../include/Trim_And_Cast.h"

#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
#include <algorithm>
#include <cstdlib>
#include <boost/tokenizer.hpp>
#include <boost/algorithm/string.hpp>

Building::Building(std::string file_name)
{
    std::ifstream input(file_name.c_str()); // initialize input stream from file: file_name
    std::string line;
    boost::char_separator<char> sep(","); // defines what separates tokens in a string
    typedef boost::tokenizer< boost::char_separator<char> > t_tokenizer; // settings for the
    boost::tokenizer

    while (!input.eof()) // continue while the End Of File has not been reached
    {
        getline(input,line); // get next line from the file
        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)
        if (line == "") //skip empty lines (tokenizer does not like it)
        {
            continue; // continue to next line
        }

        t_tokenizer tok(line, sep); // tokenize the line
        t_tokenizer::iterator token = tok.begin(); // set iterator to first token

        int room_ID;
        double x, y, z, w, d, h;

        if (*token != "R") {} // if the line starts with 'R' then a room is described by
        that line, then go to else
        else
        {
            token++;
            room_ID = trim_and_cast_int(*token);

            token++; // this is 'width'
            w = trim_and_cast_double(*token);

            token++; // this is 'depth'
            d = trim_and_cast_double(*token);

            token++; // this is 'height'
            h = trim_and_cast_double(*token);

            token++; // this is 'x-coordinate'
            x = trim_and_cast_double(*token);

            token++; // this is 'y-coordinate'
            y = trim_and_cast_double(*token);

            token++; // this is 'z-coordinate'
            z = trim_and_cast_double(*token);

            this->add_space(room_ID, x, y, z, w, d, h);
```

```
        }
    }
    //ctor
}

void Building::make_conformal()
{
    // check for intersections
    Vertex* temp_ptr = new Vertex; // to store possible intersection points
    for (unsigned int i = 0; i < m_rectangles.size(); i++)
    {
        for (unsigned int j = 0; j < m_lines.size(); j++)
        {
            if (m_rectangles[i]->check_line_intersect(m_lines[j], temp_ptr))
            {
                Vertex_Store::add_vertex(temp_ptr->get_x(), temp_ptr->get_y(),
                temp_ptr->get_z());
            }
        }
    }

    for (unsigned int i = 0; i < m_lines.size(); i++)
    {
        for (unsigned int j = 0; j < m_lines.size(); j++)
        {
            if ((i != j) && (m_lines[i]->check_line_intersect(m_lines[j], temp_ptr)))
            {
                Vertex_Store::add_vertex(temp_ptr->get_x(), temp_ptr->get_y(),
                temp_ptr->get_z());
            }
        }
    }
    delete temp_ptr;

    // check if vertices interfere with spaces
    for (unsigned int i = 0; i < m_spaces.size(); i++)
    {
        for (unsigned int j = 0; j < m_vertices.size(); j++)
        {
            m_spaces[i]->check_vertex(m_vertices[j]);
        }
    }


    // delete all lines, rectangles and cuboids that are tagged to be deleted
    for (unsigned int i = 0; i < m_cubes.size(); i++)
    {
        if (m_cubes[i]->check_deletion())
        {
            Vertex_Store::delete_cuboid(m_cubes[i]);
            i--;
        }
    }

    for (unsigned int i = 0; i < m_rectangles.size(); i++)
    {
        if (m_rectangles[i]->check_deletion())
        {
            Vertex_Store::delete_rectangle(m_rectangles[i]);
            i--;
        }
    }
```

```
    for (unsigned int i = 0; i < m_lines.size(); i++)
    {
        if (m_lines[i]->check_deletion())
        {
            Vertex_Store::delete_line(m_lines[i]);
            i--;
        }
    }

    std::ofstream out("Points.csv");
    for (unsigned int i = 0; i < m_rectangles.size(); i++)
    {
        for (int j = 0; j < 4; j++)
        {
            out << m_rectangles[i]->get_line_ptr(j)->get_vertex_ptr(0)->get_x() << "," <<
            m_rectangles[i]->get_line_ptr(j)->get_vertex_ptr(0)->get_y() << "," <<
            m_rectangles[i]->get_line_ptr(j)->get_vertex_ptr(0)->get_z() << std::endl;
            out << m_rectangles[i]->get_line_ptr(j)->get_vertex_ptr(1)->get_x() << "," <<
            m_rectangles[i]->get_line_ptr(j)->get_vertex_ptr(1)->get_y() << "," <<
            m_rectangles[i]->get_line_ptr(j)->get_vertex_ptr(1)->get_z() << std::endl;
        }
    }
    out.close();
}

void Building::make_BP_input()
{
    int counter = 0;
    int floor_count = 0;
    int wall_count = 0;
    int window_count = 0;
    for (unsigned int i = 0; i < m_spaces.size(); i++)
    {
        BP_space temp;
        temp.m_ID = m_spaces[i]->get_ID();
        temp.m_volume = (m_spaces[i]->get_encasing_cuboid().get_volume() / 1e9);
        temp.m_Q_heat = 100.0;
        temp.m_Q_cool = 100.0;
        temp.m_T_heat = 18.0;
        temp.m_T_cool = 22.0;
        temp.m_ACH = 1.0;
        m_BP_spaces.push_back(temp);
    }

    Vertex base(0,0,0);
    Vertex top(0,0,1000);
    Vector vertical(&base, &top);


    for (unsigned int i = 0; i < m_rectangles.size(); i++)
    {
        if (vertical.calc_cross_product(*m_rectangles[i]->get_normal_ptr()).is_zero())
        { //floor
            BP_floor temp;

            switch (m_rectangles[i]->get_surface_count())
            {
            case 0: // rectangle does not belong to a floor
            {
                //do nothing
                break;
```

```
            }
            case 1: // rectangle belongs to an exterior floor
            {
                floor_count++;
                if (m_rectangles[i]->get_surface_ptr(0)->get_space_count() != 1)
                {
                    std::cout << "Error, a surface was assigned to too many or too few
                    spaces (a), exiting" << std::endl;
                    exit(1);
                }
                temp.m_space_side_1 =
                std::to_string(m_rectangles[i]->get_surface_ptr(0)->get_space_ptr(0)->get_ID()
                );
                if (m_rectangles[i]->get_vertex_ptr(0)->get_z() == 0)
                {
                    temp.m_space_side_2 = "G";
                }
                else
                {
                    temp.m_space_side_2 = "E";
                }

                temp.m_area = m_rectangles[i]->get_area()/1e6;
                temp.m_ID = floor_count;
                temp.m_construction_ID = 1;
                m_BP_floors.push_back(temp);
                break;
            }
            case 2: // rectangle belongs to a separation floor
            {
                floor_count++;
                if (m_rectangles[i]->get_surface_ptr(0)->get_space_count() != 1)
                {
                    std::cout << "Error, a surface was assigned to too many or too few
                    spaces (b), exiting" << std::endl;
                    exit(1);
                }
                temp.m_space_side_1 =
                std::to_string(m_rectangles[i]->get_surface_ptr(0)->get_space_ptr(0)->get_ID()
                );

                if (m_rectangles[i]->get_surface_ptr(1)->get_space_count() != 1)
                {
                    std::cout << "Error, a surface was assigned to too many or too few
                    spaces (c), exiting" << std::endl;
                    exit(1);
                }
                temp.m_space_side_2 =
                std::to_string(m_rectangles[i]->get_surface_ptr(1)->get_space_ptr(0)->get_ID()
                );
                temp.m_area = m_rectangles[i]->get_area()/1e6;
                temp.m_ID = floor_count;
                temp.m_construction_ID = 1;
                m_BP_floors.push_back(temp);
                break;
            }
            default:
                std::cout << "Error, too many surfaces assigned to rectangle, exiting..." <<
                std::endl;
                exit(1);
                break;
            }
```

```
        }
        else
        { //wall or window

            if (counter % 3 == 0) // every third that is non separating is a window
            {
                BP_window temp_window;
                BP_wall temp_wall;

                switch (m_rectangles[i]->get_surface_count())
                {
                case 0: // rectangle does not belong to a wall
                {
                    //do nothing
                    break;
                }
                case 1: // rectangle belongs to an exterior wall
                {
                    window_count++;
                    if (m_rectangles[i]->get_surface_ptr(0)->get_space_count() != 1)
                    {
                        std::cout << "Error, a surface was assigned to too many or too few
                        spaces (a), exiting" << std::endl;
                        exit(1);
                    }
                    temp_window.m_space_side_1 =
                    std::to_string(m_rectangles[i]->get_surface_ptr(0)->get_space_ptr(0)->get_
                    ID());
                    temp_window.m_space_side_2 = "E";
                    temp_window.m_area = m_rectangles[i]->get_area()/1e6;
                    temp_window.m_ID = window_count;
                    temp_window.m_orientation = 0;
                    temp_window.m_U_value = 1.2;
                    temp_window.m_capacitance_per_area = 30000;
                    m_BP_windows.push_back(temp_window);
                    break;
                }
                case 2: // rectangle belongs to a separation wall
                {
                    wall_count++;
                    if (m_rectangles[i]->get_surface_ptr(0)->get_space_count() != 1)
                    {
                        std::cout << "Error, a surface was assigned to too many or too few
                        spaces (b), exiting" << std::endl;
                        exit(1);
                    }
                    temp_wall.m_space_side_1 =
                    std::to_string(m_rectangles[i]->get_surface_ptr(0)->get_space_ptr(0)->get_
                    ID());

                    if (m_rectangles[i]->get_surface_ptr(1)->get_space_count() != 1)
                    {
                        std::cout << "Error, a surface was assigned to too many or too few
                        spaces (c), exiting" << std::endl;
                        exit(1);
                    }
                    temp_wall.m_space_side_2 =
                    std::to_string(m_rectangles[i]->get_surface_ptr(1)->get_space_ptr(0)->get_
                    ID());
                    temp_wall.m_area = m_rectangles[i]->get_area()/1e6;
                    temp_wall.m_ID = wall_count;
```

```
                temp_wall.m_orientation = 0;
                temp_wall.m_construction_ID = 1;
                m_BP_walls.push_back(temp_wall);
                break;
            }
            default:
                std::cout << "Error, too many surfaces assigned to rectangle,
                exiting..." << std::endl;
                exit(1);
                break;
            }
        }
        else
        {
            BP_wall temp;

            switch (m_rectangles[i]->get_surface_count())
            {
            case 0: // rectangle does not belong to a wall
            {
                //do nothing
                break;
            }
            case 1: // rectangle belongs to an exterior wall
            {
                wall_count++;
                if (m_rectangles[i]->get_surface_ptr(0)->get_space_count() != 1)
                {
                    std::cout << "Error, a surface was assigned to too many or too few
                    spaces (a), exiting" << std::endl;
                    exit(1);
                }
                temp.m_space_side_1 =
                std::to_string(m_rectangles[i]->get_surface_ptr(0)->get_space_ptr(0)->get_
                ID());
                temp.m_space_side_2 = "E";
                temp.m_area = m_rectangles[i]->get_area()/1e6;
                temp.m_ID = wall_count;
                temp.m_orientation = 0;
                temp.m_construction_ID = 1;
                m_BP_walls.push_back(temp);
                break;
            }
            case 2: // rectangle belongs to a separation wall
            {
                wall_count++;
                if (m_rectangles[i]->get_surface_ptr(0)->get_space_count() != 1)
                {
                    std::cout << "Error, a surface was assigned to too many or too few
                    spaces (b), exiting" << std::endl;
                    exit(1);
                }
                temp.m_space_side_1 =
                std::to_string(m_rectangles[i]->get_surface_ptr(0)->get_space_ptr(0)->get_
                ID());

                if (m_rectangles[i]->get_surface_ptr(1)->get_space_count() != 1)
                {
                    std::cout << "Error, a surface was assigned to too many or too few
                    spaces (c), exiting" << std::endl;
                    exit(1);
                }
```

```
                    temp.m_space_side_2 =
                    std::to_string(m_rectangles[i]->get_surface_ptr(1)->get_space_ptr(0)->get_
                    ID());
                    temp.m_area = m_rectangles[i]->get_area()/1e6;
                    temp.m_ID = wall_count;
                    temp.m_orientation = 0;
                    temp.m_construction_ID = 1;
                    m_BP_walls.push_back(temp);
                    break;
                }
                default:
                    std::cout << "Error, too many surfaces assigned to rectangle,
                    exiting..." << std::endl;
                    exit(1);
                    break;
                }

            }

            counter++;
        }

    }
}

void Building::write_blocks(std::string file_name)
{
    std::ofstream blocks(file_name.c_str());

    if (!blocks.is_open())
    {
        std::cout << "Could not make or open the blocks file, exiting.." << std::endl;
        exit(1);
    }

    for (unsigned int i = 0; i < m_cubes.size(); i++)
    {
        Vertex* origin = m_cubes[i]->get_min_vertex();
        Vertex* coords = m_cubes[i]->get_max_vertex();

        blocks << "R," << m_cubes[i]->get_last_space_ptr()->get_ID() << ","
               << coords->get_x() - origin->get_x() << ","
               << coords->get_y() - origin->get_y() << ","
               << coords->get_z() - origin->get_z() << ","
               << origin->get_x() << ","
               << origin->get_y() << ","
               << origin->get_z() << std::endl;
    }

    blocks.close();
}

void Building::write_BP_input(std::string file_name)
{
    std::ofstream BP_input(file_name.c_str());

    if (!BP_input.is_open())
    {
        std::cout << "Could not make or open the BP_input file, exiting.." << std::endl;
        exit(1);
    }
```

```
    BP_input << "#Spaces,\t"
             << "Space_ID,\t"
             << "Volume [m³],\t"
             << "Heating [W/m³],\t"
             << "Cooling [W/m³],\t"
             << "Heat set point,\t"
             << "Cool set point,\t"
             << "Air changes/hour" << std::endl;
    for (unsigned int i = 0; i < m_BP_spaces.size(); i++)
    {
        BP_input << "S,\t\t"
                 << m_BP_spaces[i].m_ID << ",\t\t"
                 << m_BP_spaces[i].m_volume << ",\t\t"
                 << m_BP_spaces[i].m_Q_heat << ",\t\t"
                 << m_BP_spaces[i].m_Q_cool << ",\t\t"
                 << m_BP_spaces[i].m_T_heat << ",\t\t"
                 << m_BP_spaces[i].m_T_cool << ",\t\t"
                 << m_BP_spaces[i].m_ACH << std::endl;
    }

    BP_input << std::endl;
    BP_input << "#Walls,\t\t"
             << "Wall_ID,\t"
             << "Construction,\t"
             << "Area [m²],\t"
             << "Orientation,\t"
             << "Space side one,\t"
             << "Space side two" << std::endl;
    for (unsigned int i = 0; i < m_BP_walls.size(); i++)
    {
        BP_input << "W,\t\t"
                 << m_BP_walls[i].m_ID << ",\t\t"
                 << m_BP_walls[i].m_construction_ID << ",\t\t"
                 << m_BP_walls[i].m_area << ",\t\t"
                 << m_BP_walls[i].m_orientation << ",\t\t"
                 << m_BP_walls[i].m_space_side_1 << ",\t\t"
                 << m_BP_walls[i].m_space_side_2 << std::endl;
    }

    BP_input << std::endl;
    BP_input << "#Windows,\t"
             << "Window_ID,\t"
             << "Area [m²],\t"
             << "U-value,\t"
             << "Cap./Area,\t"
             << "Orientation,\t"
             << "Space side one,\t"
             << "Space side two" << std::endl;
    for (unsigned int i = 0; i < m_BP_windows.size(); i++)
    {
        BP_input << "V,\t\t"
                 << m_BP_windows[i].m_ID << ",\t\t"
                 << m_BP_windows[i].m_area << ",\t\t"
                 << m_BP_windows[i].m_U_value << ",\t\t"
                 << m_BP_windows[i].m_capacitance_per_area << ",\t\t"
                 << m_BP_windows[i].m_orientation << ",\t\t"
                 << m_BP_windows[i].m_space_side_1 << ",\t\t"
                 << m_BP_windows[i].m_space_side_2 << std::endl;
    }

    BP_input << std::endl;
    BP_input << "#Floors,\t"
```

```
                << "Floor_ID,\t"
                << "Construction,\t"
                << "Area,\t"
                << "Space side one,\t"
                << "Space side two" << std::endl;
    for (unsigned int i = 0; i < m_BP_floors.size(); i++)
    {
        BP_input << "F,\t\t"
                 << m_BP_floors[i].m_ID << ",\t\t"
                 << m_BP_floors[i].m_construction_ID << ",\t\t"
                 << m_BP_floors[i].m_area << ",\t\t"
                 << m_BP_floors[i].m_space_side_1 << ",\t\t"
                 << m_BP_floors[i].m_space_side_2 << std::endl;
    }

    BP_input.close();
}

Building::~Building()
{
    for (unsigned int i = 0; i < m_points.size(); i++)
    {
        delete m_points[i];
        m_points.clear();
    }

    for (unsigned int i = 0; i < m_edges.size(); i++)
    {
        delete m_edges[i];
        m_edges.clear();
    }

    for (unsigned int i = 0; i < m_surfaces.size(); i++)
    {
        delete m_surfaces[i];
        m_surfaces.clear();
    }

    for (unsigned int i = 0; i < m_spaces.size(); i++)
    {
        delete m_spaces[i];
        m_spaces.clear();
    }
    //dtor
}

Point* Building::add_point(Vertex* vertex_ptr)
{
    m_points.push_back(new Point(vertex_ptr));
    return m_points.back();
}

void Building::add_point(double x, double y, double z)
{
    Vertex* v_1 = Vertex_Store::add_vertex(x, y, z);
    Point* p_1 = new Point(v_1);
    m_points.push_back(p_1);
    v_1->add_point(p_1);
}

Edge* Building::add_edge(Line* line_ptr, Point* p_1, Point* p_2)
{
```

```
    m_edges.push_back(new Edge(line_ptr, p_1, p_2));
    return m_edges.back();
}

void Building::add_edge(double x1, double y1, double z1, double x2, double y2, double z2)
{
    Vertex* v_1, * v_2;
    Point* p_1, * p_2;
    if (((x1 == x2) && (y1 == y2)) && (z1 != z2))
    {
        v_1 = Vertex_Store::add_vertex(x1,     y1,      z1   );    p_1 = add_point(v_1);
        v_1->add_point(p_1);
        v_2 = Vertex_Store::add_vertex(x1,     y1,      z2   );    p_2 = add_point(v_2);
        v_2->add_point(p_2);
    }
    else if((x1 != x2) && ((y1 == y2) && (z1 == z1)))
    {
        v_1 = Vertex_Store::add_vertex(x1,     y1,      z1   );    p_1 = add_point(v_1);
        v_1->add_point(p_1);
        v_2 = Vertex_Store::add_vertex(x2,     y1,      z1   );    p_2 = add_point(v_2);
        v_2->add_point(p_2);
    }
    else if ((y1 == y2) && ((x1 != x2) && (z1 == z1)))
    {
        v_1 = Vertex_Store::add_vertex(x1,     y1,      z1   );    p_1 = add_point(v_1);
        v_1->add_point(p_1);
        v_2 = Vertex_Store::add_vertex(x1,     y2,      z1   );    p_2 = add_point(v_2);
        v_2->add_point(p_2);
    }
    else
    {
        std::cout << "Error, edge is not defined orthogonal, exiting..." << std::endl;
        exit(1);
    }

    Line* l_1 = Vertex_Store::add_line(v_1, v_2);
    Edge* e_1 = new Edge(l_1, p_1, p_2);
    m_edges.push_back(e_1);
    l_1->add_edge(e_1);

    p_1->add_edge(e_1); p_2->add_edge(e_1);
}

Surface* Building::add_surface(Rectangle* rectangle_ptr, Edge* e_1, Edge* e_2, Edge* e_3,
Edge* e_4)
{
    m_surfaces.push_back(new Surface(rectangle_ptr, e_1, e_2, e_3, e_4));
    return m_surfaces.back();
}

void Building::add_surface(double x1, double y1, double z1, double x2, double y2, double z2)
{
    Vertex* v_1, * v_2, * v_3, * v_4;
    Point* p_1, * p_2, * p_3, * p_4;
    if ((x1 == x2) && ((y1 != y2) && (z1 != z2)))
    {
        v_1 = Vertex_Store::add_vertex(x1,     y1,      z1  );    p_1 = add_point(v_1);
        v_1->add_point(p_1);
        v_2 = Vertex_Store::add_vertex(x1,     y1,      z2  );    p_2 = add_point(v_2);
        v_2->add_point(p_2);
        v_3 = Vertex_Store::add_vertex(x1,     y2,      z1  );    p_3 = add_point(v_3);
        v_3->add_point(p_3);
```

```
        v_4 = Vertex_Store::add_vertex(x1,     y2,      z2  );    p_4 = add_point(v_4);
        v_4->add_point(p_4);
    }
    else if ((y1 == y2) && ((x1 != x2) && (z1 != z2)))
    {
        v_1 = Vertex_Store::add_vertex(x1,     y1,      z1  );    p_1 = add_point(v_1);
        v_1->add_point(p_1);
        v_2 = Vertex_Store::add_vertex(x1,     y1,      z2  );    p_2 = add_point(v_2);
        v_2->add_point(p_2);
        v_3 = Vertex_Store::add_vertex(x2,     y1,      z1  );    p_3 = add_point(v_3);
        v_3->add_point(p_3);
        v_4 = Vertex_Store::add_vertex(x2,     y1,      z2  );    p_4 = add_point(v_4);
        v_4->add_point(p_4);
    }
    else if ((z1 == z2) && ((x1 != x2) && (y1 != y2)))
    {
        v_1 = Vertex_Store::add_vertex(x1,     y1,      z1  );    p_1 = add_point(v_1);
        v_1->add_point(p_1);
        v_2 = Vertex_Store::add_vertex(x1,     y2,      z1  );    p_2 = add_point(v_2);
        v_2->add_point(p_2);
        v_3 = Vertex_Store::add_vertex(x2,     y1,      z1  );    p_3 = add_point(v_3);
        v_3->add_point(p_3);
        v_4 = Vertex_Store::add_vertex(x2,     y2,      z1  );    p_4 = add_point(v_4);
        v_4->add_point(p_4);
    }
    else
    {
        std::cout << "Error, surface is not an orthogonal plane, exiting..." << std::endl;
        exit(1);
    }

    Line* l_01 = Vertex_Store::add_line(v_1,  v_2);     Edge* e_01 = add_edge(l_01, p_1,
    p_2);    l_01->add_edge(e_01);   p_1->add_edge(e_01);    p_2->add_edge(e_01);
    Line* l_02 = Vertex_Store::add_line(v_2,  v_4);     Edge* e_02 = add_edge(l_02, p_2,
    p_4);    l_02->add_edge(e_02);   p_2->add_edge(e_01);    p_4->add_edge(e_01);
    Line* l_03 = Vertex_Store::add_line(v_4,  v_3);     Edge* e_03 = add_edge(l_03, p_4,
    p_3);    l_03->add_edge(e_03);   p_4->add_edge(e_01);    p_3->add_edge(e_01);
    Line* l_04 = Vertex_Store::add_line(v_3,  v_1);     Edge* e_04 = add_edge(l_04, p_3,
    p_1);    l_04->add_edge(e_04);   p_3->add_edge(e_01);    p_1->add_edge(e_01);

    Rectangle* r_1 = Vertex_Store::add_rectangle(l_01,  l_02,   l_03,   l_04);
    Surface* s_1 = new Surface(r_1, e_01,  e_02,   e_03,   e_04);
    m_surfaces.push_back(s_1);
    r_1->add_surface(s_1);

    e_01->add_surface(s_1); e_02->add_surface(s_1); e_03->add_surface(s_1);
    e_04->add_surface(s_1);
    p_1->add_surface(s_1);  p_2->add_surface(s_1);  p_3->add_surface(s_1);
    p_4->add_surface(s_1);
}

void Building::add_space(int room_ID ,double x, double y, double z, double w, double d,
double h)
{
    Vertex* v_1 = Vertex_Store::add_vertex(x,     y,      z   );    Point* p_1 =
    add_point(v_1);    v_1->add_point(p_1);
    Vertex* v_2 = Vertex_Store::add_vertex(x,     y,      z+h );    Point* p_2 =
    add_point(v_2);    v_2->add_point(p_2);
    Vertex* v_3 = Vertex_Store::add_vertex(x,     y+d,    z   );    Point* p_3 =
    add_point(v_3);    v_3->add_point(p_3);
    Vertex* v_4 = Vertex_Store::add_vertex(x,     y+d,    z+h );    Point* p_4 =
    add_point(v_4);    v_4->add_point(p_4);
```

```
Vertex* v_5 = Vertex_Store::add_vertex(x+w,   y,      z   );    Point* p_5 =
add_point(v_5);    v_5->add_point(p_5);
Vertex* v_6 = Vertex_Store::add_vertex(x+w,   y,      z+h );    Point* p_6 =
add_point(v_6);    v_6->add_point(p_6);
Vertex* v_7 = Vertex_Store::add_vertex(x+w,   y+d,    z   );    Point* p_7 =
add_point(v_7);    v_7->add_point(p_7);
Vertex* v_8 = Vertex_Store::add_vertex(x+w,   y+d,    z+h );    Point* p_8 =
add_point(v_8);    v_8->add_point(p_8);


Line* l_01 = Vertex_Store::add_line(v_1,  v_3);     Edge* e_01 = add_edge(l_01, p_1,
p_3);    l_01->add_edge(e_01);   p_1->add_edge(e_01);    p_3->add_edge(e_01);
Line* l_02 = Vertex_Store::add_line(v_3,  v_7);     Edge* e_02 = add_edge(l_02, p_3,
p_7);    l_02->add_edge(e_02);   p_3->add_edge(e_02);    p_7->add_edge(e_02);
Line* l_03 = Vertex_Store::add_line(v_7,  v_5);     Edge* e_03 = add_edge(l_03, p_7,
p_5);    l_03->add_edge(e_03);   p_7->add_edge(e_03);    p_5->add_edge(e_03);
Line* l_04 = Vertex_Store::add_line(v_5,  v_1);     Edge* e_04 = add_edge(l_04, p_5,
p_1);    l_04->add_edge(e_04);   p_5->add_edge(e_04);    p_1->add_edge(e_04);
Line* l_05 = Vertex_Store::add_line(v_2,  v_4);     Edge* e_05 = add_edge(l_05, p_2,
p_4);    l_05->add_edge(e_05);   p_2->add_edge(e_05);    p_4->add_edge(e_05);
Line* l_06 = Vertex_Store::add_line(v_4,  v_8);     Edge* e_06 = add_edge(l_06, p_4,
p_8);    l_06->add_edge(e_06);   p_4->add_edge(e_06);    p_8->add_edge(e_06);
Line* l_07 = Vertex_Store::add_line(v_8,  v_6);     Edge* e_07 = add_edge(l_07, p_8,
p_6);    l_07->add_edge(e_07);   p_8->add_edge(e_07);    p_6->add_edge(e_07);
Line* l_08 = Vertex_Store::add_line(v_6,  v_2);     Edge* e_08 = add_edge(l_08, p_6,
p_2);    l_08->add_edge(e_08);   p_6->add_edge(e_08);    p_2->add_edge(e_08);
Line* l_09 = Vertex_Store::add_line(v_1,  v_2);     Edge* e_09 = add_edge(l_09, p_1,
p_2);    l_09->add_edge(e_09);   p_1->add_edge(e_09);    p_2->add_edge(e_09);
Line* l_10 = Vertex_Store::add_line(v_3,  v_4);     Edge* e_10 = add_edge(l_10, p_3,
p_4);    l_10->add_edge(e_10);   p_3->add_edge(e_10);    p_4->add_edge(e_10);
Line* l_11 = Vertex_Store::add_line(v_7,  v_8);     Edge* e_11 = add_edge(l_11, p_7,
p_8);    l_11->add_edge(e_11);   p_7->add_edge(e_11);    p_8->add_edge(e_11);
Line* l_12 = Vertex_Store::add_line(v_5,  v_6);     Edge* e_12 = add_edge(l_12, p_5,
p_6);    l_12->add_edge(e_12);   p_5->add_edge(e_12);    p_6->add_edge(e_12);


Rectangle* r_1 = Vertex_Store::add_rectangle(l_01,  l_02,   l_03,   l_04);      Surface*
s_1 = add_surface(r_1, e_01,  e_02,   e_03,   e_04);   r_1->add_surface(s_1);
Rectangle* r_2 = Vertex_Store::add_rectangle(l_05,  l_06,   l_07,   l_08);      Surface*
s_2 = add_surface(r_2, e_05,  e_06,   e_07,   e_08);   r_2->add_surface(s_2);
Rectangle* r_3 = Vertex_Store::add_rectangle(l_10,  l_05,   l_09,   l_01);      Surface*
s_3 = add_surface(r_3, e_10,  e_05,   e_09,   e_01);   r_3->add_surface(s_3);
Rectangle* r_4 = Vertex_Store::add_rectangle(l_11,  l_06,   l_10,   l_02);      Surface*
s_4 = add_surface(r_4, e_11,  e_06,   e_10,   e_02);   r_4->add_surface(s_4);
Rectangle* r_5 = Vertex_Store::add_rectangle(l_12,  l_07,   l_11,   l_03);      Surface*
s_5 = add_surface(r_5, e_12,  e_07,   e_11,   e_03);   r_5->add_surface(s_5);
Rectangle* r_6 = Vertex_Store::add_rectangle(l_09,  l_08,   l_12,   l_04);      Surface*
s_6 = add_surface(r_6, e_09,  e_08,   e_12,   e_04);   r_6->add_surface(s_6);

e_01->add_surface(s_1); e_02->add_surface(s_1); e_03->add_surface(s_1);
e_04->add_surface(s_1); p_1->add_surface(s_1);  p_3->add_surface(s_1);
p_5->add_surface(s_1);  p_7->add_surface(s_1);
e_05->add_surface(s_2); e_06->add_surface(s_2); e_07->add_surface(s_2);
e_08->add_surface(s_2); p_2->add_surface(s_2);  p_4->add_surface(s_2);
p_8->add_surface(s_2);  p_6->add_surface(s_2);
e_10->add_surface(s_3); e_05->add_surface(s_3); e_09->add_surface(s_3);
e_01->add_surface(s_3); p_1->add_surface(s_3);  p_3->add_surface(s_3);
p_4->add_surface(s_3);  p_2->add_surface(s_3);
e_11->add_surface(s_4); e_06->add_surface(s_4); e_10->add_surface(s_4);
e_02->add_surface(s_4); p_7->add_surface(s_4);  p_8->add_surface(s_4);
p_4->add_surface(s_4);  p_3->add_surface(s_4);
e_12->add_surface(s_5); e_07->add_surface(s_5); e_11->add_surface(s_5);
```

```
    e_03->add_surface(s_5); p_5->add_surface(s_5);  p_6->add_surface(s_5);
    p_8->add_surface(s_5);  p_7->add_surface(s_5);
    e_09->add_surface(s_6); e_08->add_surface(s_6); e_12->add_surface(s_6);
    e_04->add_surface(s_6); p_1->add_surface(s_6);  p_2->add_surface(s_6);
    p_5->add_surface(s_6);  p_6->add_surface(s_6);


    Cuboid* c_1 = Vertex_Store::add_cuboid(r_1,   r_2,    r_3,    r_4,    r_5,    r_6);
    Space* space_1 = new Space(room_ID, c_1, s_1,   s_2,    s_3,    s_4,    s_5,    s_6);
    m_spaces.push_back(space_1);
    c_1->add_space(space_1);

    p_1->add_space(space_1);    p_2->add_space(space_1);    p_3->add_space(space_1);
    p_4->add_space(space_1);
    p_5->add_space(space_1);    p_6->add_space(space_1);    p_7->add_space(space_1);
    p_8->add_space(space_1);

    e_01->add_space(space_1);   e_02->add_space(space_1);   e_03->add_space(space_1);
    e_04->add_space(space_1);   e_05->add_space(space_1);   e_06->add_space(space_1);
    e_07->add_space(space_1);   e_08->add_space(space_1);   e_09->add_space(space_1);
    e_10->add_space(space_1);   e_11->add_space(space_1);   e_12->add_space(space_1);

    s_1->add_space(space_1);    s_2->add_space(space_1);    s_3->add_space(space_1);
    s_4->add_space(space_1);    s_5->add_space(space_1);    s_6->add_space(space_1);
}

Surface* Building::get_surface(int n)
{
    return m_surfaces[n];
}

Space* Building::get_space(int n)
{
    return m_spaces[n];
}

Cuboid* Building::get_cuboid(int n)
{
    return Vertex_Store::m_cubes[n];
}
Rectangle* Building::get_rectangle(int n)
{
    return Vertex_Store::m_rectangles[n];
}

Line* Building::get_line(int n)
{
    return Vertex_Store::m_lines[n];
}
```