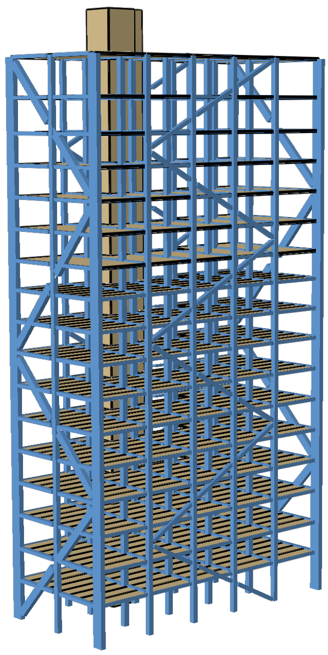
Daniel Hjohlman Reed
Lars Håkon Wiig

A Parametric Study of Tall Timber Buildings

Master's thesis in Civil and Environmental Engineering
Supervisor: Kjell Arne Malo
June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Structural Engineering



NTNU
Norwegian University of
Science and Technology

Daniel Hjohlman Reed
Lars Håkon Wiig

A Parametric Study of Tall Timber Buildings

Master's thesis in Civil and Environmental Engineering
Supervisor: Kjell Arne Malo
June 2020

Norwegian University of Science and Technology
Faculty of Engineering
Department of Structural Engineering

NTNU
Norwegian University of
Science and Technology

Department of Structural Engineering
Faculty of Engineering
NTNU – Norwegian University of Science and Technology

ACCESSIBILITY:
Open

MASTER THESIS 2020

SUBJECT AREA:	DATE:	NO. OF PAGES:
Timber Structures	**June 10, 2020**	**125 (Thesis) + 204 (Appendix)**

TITLE:

A Parametric Study of Tall Timber Buildings

En parametrisk studie av høye trehus

BY:

Daniel Hjohlman Reed
Lars Håkon Wiig


SUMMARY:
DynaTTB is a research project dedicated to the response of tall timber buildings under service loads. One of the objectives for the project is to identify the effects of different stiffness, mass and damping parameters on the dynamic response of the structure. The main part of this thesis has been dedicated towards the development of a parametric finite element model. The model is programmed in Python and is intended for use with the finite element software Abaqus. The model offers a variety of different parameters related to geometry, mass, stiffness and damping of both the foundation, structural members, connections etc. All of which are gathered in a Microsoft Excel file to make the setup user-friendly.

The parametric model is then used to model Mjøstårnet, which at 85.4 m is the tallest timber building in the world. A sensitivity study is conducted where the sensitivity of the three fundamental frequencies for changes in a variety of different stiffness parameters is measured. Some of most important parameters are found to be the vertical foundation stiffness, axial stiffness of connections in the bracing system and the stiffness of the exterior wall panels. On the other hand, the stiffness of the floors and the rotational stiffness of the foundations are among the parameters found to be relatively unimportant.

The parameters found to be most important in the sensitivity study are then included in a simple model updating routine where the aim is to find the values of the parameters that yields the same model output as measured in real life. Three different runs are presented and the results are discussed.

Finally, the updated model of Mjøstårnet is used to demonstrate the capabilities of the script to perform wind load analyses after the Eurocode. A parameter study is performed where different damping and wind-related parameters are modified and the acceleration response is studied. The results are compared with on-site measurements and recommended threshold values.

RESPONSIBLE TEACHER: Kjell Arne Malo

SUPERVISOR(S): Kjell Arne Malo

CARRIED OUT AT: Department of Structural Engineering, NTNU

Preface

This thesis concludes our master studies in *Civil and Environmental Engineering* at NTNU in Trondheim. The thesis is written for the Timber structures group at the Department of Structural Engineering, and is a part of the research project *Dynamic Response of Tall Timber Buildings under Service Load (DynaTTB)*.

We have been fortunate enough to be given a masters project by our supervisor prof. Kjell Arne Malo that has allowed us to work with several interesting topics. We have been allowed to shape the thesis after our interest in timber structures, structural dynamics and finite element modelling. In addition we have gained new and valuable experience with development of scripts in Python.

We would like to express our sincere gratitude to our supervisor prof. Kjell Arne Malo at the Department of Structural Engineering for providing valuable guidance in the work with this thesis. We would also like to thank Sweco for providing access to drawings and models of Mjøstårnet, and PhD-Candidate Saule Tulebekova for giving us access to her work on the data from the on-site measurements. Finally, we would like to express our gratefullness to the all the member of the Timber structure group at NTNU for providing advise and support when needed.

Daniel Hjohlman Reed & Lars Håkon Wiig
Trondheim, June 2020

Abstract

DynaTTB is a research project dedicated to the response of tall timber buildings under service loads. One of the objectives for the project is to identify the effects of different stiffness, mass and damping parameters on the dynamic response of the structure. The main part of this thesis has been dedicated towards the development of a parametric finite element model. The model is programmed in Python and is intended for use with the finite element software Abaqus. The model offers a variety of different parameters related to geometry, mass, stiffness and damping of both the foundation, structural members, connections etc. All of which are gathered in a Microsoft Excel file to make the setup user-friendly.

The parametric model is then used to model Mjøstårnet, which at 85.4 m is the tallest timber building in the world. A sensitivity study is conducted where the sensitivity of the three fundamental frequencies for changes in a variety of different stiffness parameters is measured. Some of most important parameters are found to be the vertical foundation stiffness, axial stiffness of connections in the bracing system and the stiffness of the exterior wall panels. On the other hand, the stiffness of the floors and the rotational stiffness of the foundations are among the parameters found to be relatively unimportant.

The parameters found to be most important in the sensitivity study are then included in a simple model updating routine where the aim is to find the values of the parameters that yields the same model output as measured in real life. Three different runs are presented and the results are discussed.

Finally, the updated model of Mjøstårnet is used to demonstrate the capabilities of the script to perform wind load analyses after the Eurocode. A parameter study is performed where different damping and wind-related parameters are modified and the acceleration response is studied. The results are compared with on-site measurements and recommended threshold values.

Sammendrag

DynaTTB er et forskningsprosjekt som fokuserer på den dynamiske responsen til høye trehus påvirket av laster i bruksgrensetilstanden. Et av målene til prosjektet er å identifisere effekten ulike stivhets-, masse- og dempningsparametere har på den dynamiske responsen av bygget. Det meste av arbeidet med oppgaven har vært dedikert til utvikling av en parametrisk elementmodell. Modellen er programmert i Python og brukes med elementprogrammet Abaqus. Modellen er definert av mange ulike parametere relatert til geometri, masse, stivhet og demping i både fundament, konstruksjonsdeler, knutepunkt osv. Alle disse parameterne er samlet i en Microsoft Excel fil for å gjøre oppsettet av modellen brukervennlig.

Den parametriske modellen er deretter brukt for å modellere Mjøstårnet, som med en høyde på 85.4 m er verdens høyeste trehus. En sensitivitetsstudie er gjennomført der sensitiviteten til de tre fundamentale frekvensene for endringer i et utvalg stivhetsparametere er målt. Noen av parameterne som viser seg å ha størst påvirkning er den vertikale stivheten til fundamentene, den aksielle stivheten til knutepunkt i de diagonale avstiverne og stivheten i de ytre veggelementene. På den andre siden er stivheten til gulvdekkene og rotasjonsstivheten til fundamentene blant parameterne som viser seg å være relativt uviktige.

Parameterne som sensitivitetsstudien viser at er viktigst blir deretter inkludert i en enkel modelloppdateringsprosedyre, der målet er å finne verdier av parameterne som gir like resultater som målingene av det eksisterende bygget. Tre ulike gjennomkjøringer er presentert og resultatene er diskutert.

Til slutt er den oppdaterte modellen av Mjøstårnet brukt til å demonstrere mulighetene det parametriske scriptet har for å utføre vindlastanalyser i henhold til Eurokoden. En parameterstudie er utført der ulike demping og vindrelaterte parametere er modifisert. Videre er den resulterende akselerasjonen studert. Resultatene er sammenlignet med målinger og anbefalte grenseverdier.

Contents

Nomenclature

Abbreviations

AER	Annual Exceedance Probability
CAE	Complete Abaqus Environment
CEN	Comité Européen de Normalisation
CLT	Cross Laminated Timber
DD-SSI	Data-Driven Stochastic Subspace Identification
DOF	Degree of Freedom
EC	Eurocode
EOM	Equation of Motion
FEA	Finite Element Analysis
Glulam	Glued Laminated timber
GUI	Graphical User Interface
ISO	International Organization for Standardization
LVL	Laminated Veneer Lumber
MAC	Modal Assurance Criterion
MDOF	Multiple Degree of Freedom
RMS	Root Mean Square
SDOF	Single Degree of Freedom
SLS	Serviceability Limit State
ULS	Ultimate Limit State

Symbols

$\boldsymbol{C}$	Damping Matrix
$\boldsymbol{K}$	Stiffness Matrix
$\boldsymbol{M}$	Mass Matrix
ν	Poisson's Ratio
ζ_i	Damping Ratio of the i^{th} Mode
A	Area
E_0	Young's Modulus Parallel to Grain
E_i	Young's Modulus in Local Material Axis i $(i = 1, 2, 3)$
E_{90}	Young's Modulus Perpendicular to Grain
f_i	The i^{th} Eigenfrequency
G_{ij}	Shear Modulus in Plane ij $(i, j = 1, 2, 3,\ i \neq j)$
I	2^{nd} Moment of Area

Chapter 1

Introduction

1.1 Background and Motivation

Fighting climate change and finding solutions to the environmental issues is becoming increasingly more important in the time to come. Reducing the emissions of greenhouse gasses is a priority for governments all over the world. As a consequence the Norwegian Government is aiming to cut emissions with 50 percent by 2030 [1]. Production and transportation of construction materials is a significant contribution to the total greenhouse gas emissions, hence using materials with lower carbon footprints is of high interest. Timber as a structural material is widely regarded to be a better choice with respect to greenhouse gas emissions than its more conventional counterparts steel and concrete. As a result, the use of structural timber in larger construction projects has gained traction. However, since the use of timber in larger structures have been limited until recent years, its properties are not as well documented as for steel and concrete. A lot of research is therefore required in order to fully substitute these materials with timber. *The Dynamic Response of Tall Timber Buildings under Service Load* or *DynaTTB* for short, is an international collaborative project focusing on the dynamic properties of timber, which this thesis is a part of. The goal of the project is the following [2]:

> "*Its aim is to quantify the structural damping in as-built tall timber buildings (TTB), identify and quantify the effects of connections and non-structural elements on the stiffness, damping and wind-induced dynamic response of TTBs, develop a bottom-up numerical finite element model for estimating the dynamic response of multi-storey timber buildings, validate the predicted response with in-situ measurements on TTBs and disseminate findings via a TTB Design Guideline for design practitioners.*"

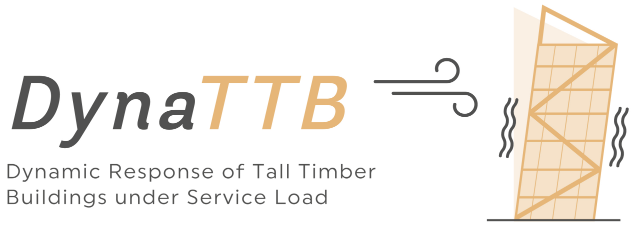


Figure 1.1: DynaTTB, from [2]

1.2 Project Description

The work done in this master thesis can be divided into two parts. The first part consists of the development of a parametric finite element model for tall timber buildings utilizing a wood-frame as the main load bearing structure. The purpose of the model is to be a tool that can be used to study how different properties influence the dynamic performance of this type of buildings. The parametric model was created by using the finite element analysis program Abaqus combined with scripting in Python. This allows the user to easily control all parameters that define the model. The majority of the work done in the thesis has been dedicated towards the development of the model.

In the second part, the parametric model is used as a tool for conducting numerical analyses. In order to study a realistic structure, the worlds tallest timber building "Mjøstårnet", with 18 stories and a total height of 81.4 m, is used as a case building. Three studies were conducted:

- A sensitivity study that examines what parameters are most important for the dynamic performance of the building.
- Based on the results from the sensitivity study and measurements of the real structure, the parameters of the initial model was altered in order to recreate the real structure as close as possible. This was done through model updating.
- Finally, a parameter study was performed to examine how different structural and wind-related parameters influence the acceleration response of a tall timber building. The results of a static wind load analysis after the method given in Eurocode 1 is also presented.

1.3 Limitations

Listed below are the main limitations for work of this thesis:

- The focus of this thesis is the dynamic properties of tall timber buildings. Evaluation of the ultimate limit state (ULS) is therefore not considered, and the parametric script that is produced should not be used to extract stresses and strains in the structure.
- The thesis only focuses on tall timber buildings using a frame as the main load-bearing system. Buildings using CLT as the main load-bearing members, can not be studied with the help of the scripts developed, in its current state.
- The option of including various kinds of damping to the model, has been a focus during the development of the parametric scripts. However, analyses conducted in this work mainly focus on how the mass and stiffness properties of the members influence the dynamic behaviour.

1.4 Outline of Thesis

A short description of the different chapter of the thesis is given in the following list:

- Background - This chapter presents the theory and background information that will be utilized in this thesis.
- Modelling - This chapter presents the parametric model that has been established for the thesis. Assumptions and simplifications made during the modelling process are discussed.
- Case Study: Mjøstårnet - This chapter explain how a preliminary model of Mjøstårnet is made using the parametric script. The model established will be used as a base model for further studies.
- Sensitivity Study - In this chapter, a study of how different parameters influence the response of the model is carried out. The parameters studied are explained, the reason for studying them are discussed and the results of the studies presented. Finally, the influence of the different parameters are compared.
- Model Updating - This chapter focuses on how the base model established in the Case Study chapter can be altered, in order to get results as close to measurements of the real building as possible.
- Wind Loads - This chapter is meant to be a demonstration of some of the

possibilities for doing wind-related analyses using the scripts that are developed in the thesis.

- Discussion - The discussion is split into two main parts. In the first part the results from all the analyses previously presented in the thesis is discussed and compared. In the second part, the capabilities of the parametric model that has been developed is discussed. Possible improvements are also presented.
- Conclusion - Finally, the conclusion of the thesis is presented along with suggestions for further work.
- Appendix A: User Guide - A description of how to set up a new parametric model.
- Appendix B: Digital Appendix - Input files, Python scripts and analysis results. Delivered directly to professor Malo at the Department of Structural Engineering at NTNU.
- Appendix C: Python Scripts - All python scripts developed for the parametric model.

Chapter 2

Background

2.1 Timber as a Structural Material

Although timber is an ancient material that has been utilized in construction for many centuries, the use of the material has historically been limited to small and relatively simple structures. Data from Finland show that there is a big distinction in the use of timber in small, private houses compared to multi-storey buildings [3], and it is reasonable to assume that the situation in Norway is similar to this. Timber has generally been restricted from use in buildings with more than two stories up until the mid 1990s due to the combustibility of the material. In 1997, new function-based fire regulations were introduced in Norway, allowing for a greater use of wood in multi-storey buildings [4]. Till this day steel and concrete have been the most widely used structural materials. However, due to the increased focus on sustainability and environmental issues, timber is becoming a more recognized material that in many cases can compete with both steel and concrete, not only for small houses, but also for larger structures.

2.1.1 Environmental Benefits of Timber

One of the major benefits of timber compared to other structural materials is its more environmentally friendly. In fact, it can be argued that this is the main reason behind the increased popularity of timber in recent years. In 2007 Bernhard and Jørgensen [5] estimated that the production of building materials is responsible for 7% of the total greenhouse gas emissions in Norway, hence choosing materials with low carbon footprint can reduce the total greenhouse gas emission significantly.

Timber is, in regards to greenhouse gas emissions, a good choice due to two main aspects: substitution and carbon storage. Selvig [6] show that if used correctly, timber as a substitute to other building materials will reduce the total CO_2-emission. However, some solutions using timber proved to be more emission heavy than the materials it was substituting. This show that careful planning is required in order to take advantage of the environmental benefits of timber. Timber is a material that naturally bonds carbon. A timber structure will during its entire lifetime store carbon, and thus reduce the amount of CO_2 in the atmosphere. The benefit of carbon storage is increased if long lifetime of the timber products and rapid regrowth after harvesting is pursued [7]. The effect of carbon storage was not included in [6].

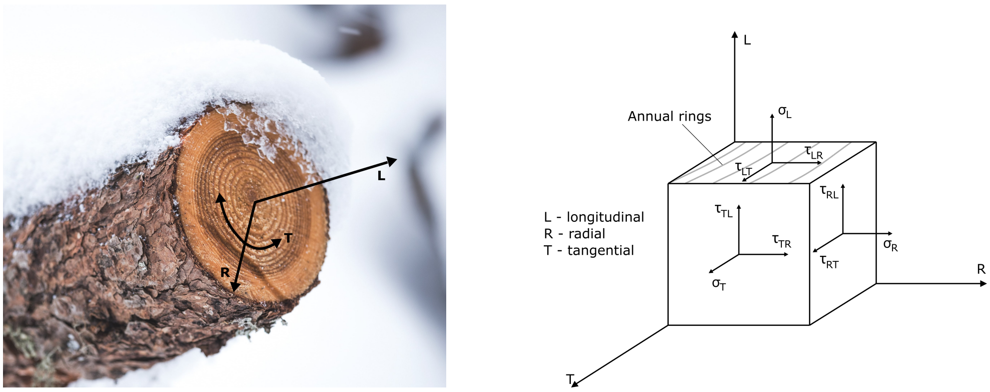
Timber have various other benefits, and some of them are listed below [7]:

- When the forestry is handled correctly, timber can be considered a renewable resource.
- Timber can be reused, both as a structural material and as an energy source.
- If designed correctly, timber structures can have a very long lifetime. The oldest timber buildings in Norway is approximately 1000 years old.
- The use of timber can improve the indoor climate.

2.1.2 Mechanical Properties of Timber

To understand the mechanical properties of timber, it is necessary to study the anatomy of wood. Wood is a natural and complex composite material with three main elements: cellulose, hemicellulose and ligning. The cellulose, a long organic chain molecule, is collected in crystalline strands, called microfibrills. These microfibrills are surrounded by the hemicellulose, a shorter chain molecule, and ligning, a generic term for a group of three dimensional polymers. The microfibrills form tube like cells, that enables water and nutrition to be transported within the tree. The cells are mainly oriented along the stem, and bound together by lignin, which act as a adhesive layer between the cells [8].

The structure of wood results in a highly anisotropic material. Three orthogonal directions are defined in order to describe the anisotropy; the **longitudinal direction, L** is the same as the longitudinal direction of the tree. The cells are oriented along this direction, and thus makes timber strongest and stiffest in this direction. The **radial direction, R,** is the direction that is perpendicular to the annual rings, while the **tangential direction, T,** is tangential to the annual rings. Timber can be compared to a reinforced material, where the cells acts as reinforcement in a matrix of lignin. The orientation of the directions are illustrated in Figure 2.1a.



(a) Material directions of wood

(b) Stress components

Figure 2.1: Definitions

Due to the anisotropic nature of wood, a three-dimensional Hooke's Law is required in order to relate stresses and strains in the material. A thorough derivation of how this relation can be established is presented by Malo [9]. The general form of Hooke's law for linear elastic materials reads

$$\boldsymbol{\sigma} = \boldsymbol{C}\boldsymbol{\epsilon} \;\; and \;\; \boldsymbol{\epsilon} = \boldsymbol{S}\boldsymbol{\sigma} \tag{2.1}$$

where **C** is the stiffness matrix and **S** is the compliance matrix. By assuming that wood have three planes of symmetry, i.e. is orthotropic, the compliance relation can be derived as:

$$\begin{bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \gamma_{23} \\ \gamma_{31} \\ \gamma_{12} \end{bmatrix} = \begin{bmatrix} \frac{1}{E_1} & \frac{-\nu_{12}}{E_1} & \frac{-\nu_{13}}{E_1} & 0 & 0 & 0 \\ & \frac{1}{E_2} & \frac{-\nu_{23}}{E_2} & 0 & 0 & 0 \\ & & \frac{1}{E_3} & 0 & 0 & 0 \\ & & & \frac{1}{G_{23}} & 0 & 0 \\ & sym. & & & \frac{1}{G_{31}} & 0 \\ & & & & & \frac{1}{G_{12}} \end{bmatrix} \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{31} \\ \sigma_{12} \end{bmatrix} \tag{2.2}$$

The stress components are defined in figure 2.1b. Note that in equation 2.2, the naming of the axes defined in figure 2.1a are substituted with numbers, such that:

$$L = 1, \;\; R = 2, \;\; T = 3$$

As seen in equation 2.2, nine independent parameters must be defined in order to model the elastic behavior of timber:

$$E_1, \; E_2, \; E_3, \; \nu_{21}, \; \nu_{31}, \; \nu_{32}, \; G_{23}, \; G_{31} \; G_{23}$$

The parameters can be determined by testing.

In practical design the difference between material properties in R- and T-directions are often neglected, as both are of similar magnitude. In addition, the designing engineer have little knowledge of how the annual rings are oriented in the finished product. Thus, a simplified transversely isotropic material model is often used. In this model, the material properties in any direction in the plane oriented perpendicular to the L-axis are considered the same. The compliance relation can then be reduced to only contain five independent parameters [9]:

$$\begin{bmatrix}\epsilon_{11}\\ \epsilon_{22}\\ \epsilon_{33}\\ \gamma_{23}\\ \gamma_{31}\\ \gamma_{12}\end{bmatrix}=\begin{bmatrix}\frac{1}{E_1} & \frac{-\nu_{12}}{E_1} & \frac{-\nu_{12}}{E_1} & 0 & 0 & 0\\ & \frac{1}{E_1} & \frac{-\nu_{23}}{E_2} & 0 & 0 & 0\\ & & \frac{1}{E_2} & 0 & 0 & 0\\ & & & \frac{2(1+\nu_{23})}{E_2} & 0 & 0\\ & sym. & & & \frac{1}{G_{12}} & 0\\ & & & & & \frac{1}{G_{12}}\end{bmatrix}\begin{bmatrix}\sigma_{11}\\ \sigma_{22}\\ \sigma_{33}\\ \sigma_{23}\\ \sigma_{31}\\ \sigma_{12}\end{bmatrix} \tag{2.3}$$

The stiffness moduli, E_2 and G_{12}, are related to deformations in the transverse plane. They are often represented as averages of the associated stiffness moduli in R- and T-direcion.

Timber Compared to Steel and Concrete

Compared to steel and concrete, timber have both low stiffness and strength. Timber is, however, a light material, which in turn results in low dead loads. It is therefore interesting to compare the ratio of strength and stiffness to weight, to get a better perception of how timber compares to steel and concrete. Specific compression strength and specific stiffness are measures that are suitable for such a comparison. They are defined as modulus of elasticity divided by density and compression strength divided by density, respectively. For the comparison, S355 structural steel, C30 concrete and C24 structural timber has been chosen as they are all widely used strength classes of the respective materials. The material properties are shown in Table 2.1, and are taken from [10], [11] and [12], respectively. The material properties of timber are for the longitudinal direction.

As Table 2.2 show, the specific strength and stiffness of timber is greater than that of concrete, and similar to that of steel. This demonstrates that timber is a material that, in many cases, can substitute the more widely used materials without sacrificing the structural performance. The low density does, however, also introduce challenges to timber structures. As timber buildings are typically lighter and more flexible compared to more conventional buildings, they tend

Table 2.1: Material Properties of Concrete, Steel and Timber

Material	Density [kg/m^3]	Compressive Strength [MPa]	Young's Modulus [MPa]
Concrete (C30)	2500	30	33 000
Steel (S355)	7800	355	210 000
Timber (C24)	420	21	11 000

Table 2.2: Specific strength and stiffness

Material	Specific Strength [Pam^3/kg]	Specific Stiffness [MPam^3/kg]
Concrete (C30)	12.0	13.2
Steel (S355)	45.5	26.9
Timber (C24)	50.0	26.2

to be susceptible for vibrations induced by human activity and wind loads. Thus, satisfying the serviceability limit state has proven to be one of main limiting factors of using timber in tall buildings.

2.1.3 Damping in Timber Structures

Timber structures does in general have a higher damping compared to structures made of steel and concrete [13]. Experimental studies show that the damping ratio for complete wood-frame shear wall systems under low level deformations is in the range of $\zeta = 0.02$ - 0.1 [14]. For higher levels of deformation, the damping ratio can be increased to as much as $\zeta = 0.2$. Typical sources for damping in wood-frame shear wall structures are material damping, friction between connected components and plastic deformations in connections. The interaction between a structure and the supporting soil is also causing energy dissipation. However, the mechanisms causing damping in timber structures are not fully understood, making it very challenging for designing engineers to predict the damping characteristics of a structure. This often lead to damping being neglected or included as a global damping ratio with unclear origins during design [13]. This was evident for the design of "Treet", a 14-storey residential building located in Bergen, Norway. The design used a total equivalent damping ratio of $\zeta = 0.019$, a value that is solely an estimation [15]. Increased knowledge on the damping properties of timber structures is important in order to be able to overcome the limitations of tall timber buildings.

2.2 Structural Dynamics

A dynamic analysis takes into account the time-dependant properties of the loading and response of a structure. The different types of time-dependant loads can be classified as random-, periodic- or impulse-loading [16]. Examples of dynamic loads are wind, people walking or running, earthquakes, waves and explosions.

The dynamic behaviour of a structure is of great importance when designing slender structures like tall buildings and long bridges. Insufficient attention to the dynamic properties of these types of structures may often lead to unwanted effects such as large accelerations and deformations. Structural dynamics is most often a serviceability issue, e.g. large accelerations causing discomfort for the residents of a tall building. However, in some extreme cases entire structures have collapsed due to dynamic loading, e.g. *The Tacoma Narrows Bridge*, which collapsed less than five months after its opening in 1940 [17]. Repeated loading and unloading due to dynamic loading may also cause fatigue issues.

2.2.1 Equation Of Motion

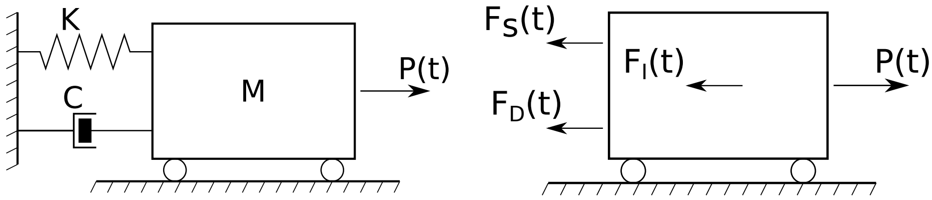
Figure 2.2a shows a simple one degree of freedom system excited by an external time-dependant force $P(t)$ consisting of a block with mass M, rolling frictionless without air resistance on a horizontal plane. The block is connected to a spring and a damper, both with negligible mass. Using *D'Alembert principle of dynamic equilibrium* [18], the free body diagram becomes as shown in Figure 2.2b, and gives the following equilibrium equation:

$$P(t) - F_S(t) - F_D(t) - F_I(t) = 0 \tag{2.4}$$

Assuming linear elastic behavior, the force in the spring is the spring stiffness K multiplied with the displacement u. The force caused by a viscous damper are equal to a coefficient C multiplied with the velocity $\dot{u}$, while Newton's second law of motion says that the inertia force equals mass M times acceleration $\ddot{u}$. Hence:

$$F_S = K \cdot u(t) \tag{2.5a}$$
$$F_D = C \cdot \dot{u}(t) \tag{2.5b}$$
$$F_I = M \cdot \ddot{u}(t) \tag{2.5c}$$

The equilibrium equation (Equation 2.4) may be rewritten using Equation 2.5,



(a) SDOF system **(b)** Forces acting on SDOF system

Figure 2.2: Simple one degree of freedom system

resulting in the *equation of motion (EOM)*:

$$M\ddot{u}(t) + C\dot{u}(t) + Ku(t) = P(t) \tag{2.6}$$

A useful modification of Equation 2.6 for free vibration ($P(t) = 0$) is to express the EOM in terms of the natural frequency ω_n and the damping ratio ζ (similar modifications may be done for harmonic and other types of loading):

$$\ddot{u} + 2\omega_n\zeta\dot{u} + \omega_n^2 u = 0 \tag{2.7}$$

where:

$$\omega_n = \sqrt{\frac{K}{M}}$$
$$\zeta = \frac{C}{C_{cr}} = \frac{C}{2M\omega_n}$$

The derivation of the EOM of a single degree of freedom system are presented above. However most structures are modeled using multiple degrees of freedom, often hundreds or even thousands of DOFs are used. The equation of motion for a system with n degrees of freedom and m time steps is written on matrix form:

$$\boldsymbol{M}\ddot{\boldsymbol{u}}(t) + \boldsymbol{C}\dot{\boldsymbol{u}}(t) + \boldsymbol{K}\boldsymbol{u}(t) = \boldsymbol{P}(t) \tag{2.8}$$

where:

$\boldsymbol{M}, \boldsymbol{C}, \boldsymbol{K}$ = System mass, damping and stiffness matrices ($n \times n$)
$\boldsymbol{P}(t)$ = System load vector ($n \times m$)
$\boldsymbol{u}(t)$ = Displacement vector ($n \times m$)
$\dot{\boldsymbol{u}}(t), \ddot{\boldsymbol{u}}(t)$ = First and second time-derivatives of the displacement ($n \times m$)

2.2.2 Modal Analysis

In general the system of equations in Equation 2.8 is coupled and complicated to solve. However it is possible to transform it such that it becomes a system of

n uncoupled equations, equivalent to n single degree of freedom systems. The transformation is explained in detail by e.g. Chopra [18], and the main steps are presented below.

Due to the relatively low damping in civil engineering structures, the damping is usually disregarded when computing the mode shapes of vibration. When damping is disregarded the mode shapes and natural frequencies become real, due to the symmetry and positive definiteness of $\boldsymbol{K}$ and $\boldsymbol{M}$ [18]. It can then be shown that the equation of motion may be rewritten as a *matrix eigenvalue problem*:

$$[\boldsymbol{K} - \omega_n^2 \boldsymbol{M}]\phi_n = 0 \tag{2.9}$$

where:

ω_n = The n^{th} natural frequency of the system (scalar)
ϕ_n = The n^{th} mode shape vector ($n \times 1$)

An important property of the mode shapes is that they can be used to orthogonalize the system, such that: $\phi_i^T \boldsymbol{K} \phi_j = 0$ and $\phi_i^T \boldsymbol{M} \phi_j = 0$ for all $i \neq j$, i.e. the stiffness and mass matrices become diagonal. Rewriting the equation system in terms of generalized degrees of freedom $\boldsymbol{q}$ simplifies the solution, the relationship between the physical DOFs $\boldsymbol{u}$ and $\boldsymbol{q}$ are as follows:

$$\boldsymbol{u}(t) = \boldsymbol{\phi} \boldsymbol{q}(t) \tag{2.10}$$

where:

$\boldsymbol{\phi}$ = A matrix where each column represent a mode shape

Substituting Equation 2.10 into the equation of motion (Equation 2.8) (still disregarding damping):

$$\boldsymbol{M}\boldsymbol{\phi}\ddot{\boldsymbol{q}}(t) + \boldsymbol{K}\boldsymbol{\phi}\boldsymbol{q}(t) = \boldsymbol{P}(t) \tag{2.11}$$

Then pre-multiply with the transposed mode shape matrix to get the transformed system:

$$\boldsymbol{M}^*\ddot{\boldsymbol{q}}(t) + \boldsymbol{K}^*\boldsymbol{q}(t) = \boldsymbol{P}^*(t) \tag{2.12}$$

where:

$\boldsymbol{M}^* = \boldsymbol{\phi}^{\boldsymbol{T}}\boldsymbol{M}\boldsymbol{\phi}$ - A square and diagonal mass matrix
$\boldsymbol{K}^* = \boldsymbol{\phi}^{\boldsymbol{T}}\boldsymbol{K}\boldsymbol{\phi}$ - A square and diagonal stiffness matrix
$\boldsymbol{P}^* = \boldsymbol{\phi}^{\boldsymbol{T}}\boldsymbol{P}$ - Load vector

Since the system is uncoupled it can be divided into many smaller SDOF-system and solved one-by-one. The EOM for each SDOF system are:

$$\boldsymbol{M}_{ii}^*\ddot{\boldsymbol{q}}_i(t) + \boldsymbol{K}_{ii}^*\boldsymbol{q}_i(t) = \boldsymbol{P}_i^*(t) \tag{2.13}$$

After each SDOF system are solved the generalized DOFs are transformed back to the original DOFs using the relation given in Equation 2.10: $\boldsymbol{u}(t) = \boldsymbol{\phi}\boldsymbol{q}(t)$. Because the response usually are dominated by the first few modes the engineer often choose the exclude the higher modes to save calculation time, however this should be done with care not to omit any important modes.

2.2.3 Damping

Even when damping is considered in a system it is common to use the mode shapes from equation (2.9) to orthogonalize the mass and stiffness matrices. However, by introducing a damping matrix the system in general becomes coupled again, i.e. $\phi_i^T \boldsymbol{C} \phi_j \neq 0$. A common solution to this problem is to construct a damping matrix that are proportional to the stiffness and mass matrices. This damping model are called *Rayleigh damping*, named after the British scientist Lord Rayleigh. The Rayleigh damping model simply defines the damping matrix $\boldsymbol{C}$ as follows [19]:

$$\boldsymbol{C} = \alpha \boldsymbol{M} + \beta \boldsymbol{K} \tag{2.14}$$

Where α and β are constants who are determined by measurements or experience from similar projects. The resulting damping ratio varies with frequency, as shown in Figure 2.3 A damping matrix defined by using Rayleigh damping may be orthogonalized just like the mass and stiffness matrices, i.e. $\phi_i^T \boldsymbol{C} \phi_j = 0 \ (i \neq j)$, and the system can be solved as multiple SDOF systems.

$$\boldsymbol{M}_{ii}^{*} \ddot{\boldsymbol{q}}_i(t) + \boldsymbol{C}_{ii}^{*} \dot{\boldsymbol{q}}_i(t) + \boldsymbol{K}_{ii}^{*} \boldsymbol{q}_i(t) = \boldsymbol{P}_i^{*}(t) \tag{2.15}$$

An alternative to defining the system damping matrix $\boldsymbol{C}$ is to introduce modal damping ratios, ζ_i, directly in to the rewritten EOM, ref. Equation 2.7, for each relevant mode. As with Rayleigh damping the value of the damping parameter is determined by experiments or engineering judgement.

One problem with methods like Rayleigh and modal damping is that they lack physical meaning, they are just applied because they are convenient and makes the system easy to solve. It doesn't say anything about what is causing the damping, and that makes it difficult to get an accurate estimate of parameters to be used in the modelling of new structures. To be able to make more accurate dynamic models of structures, more complicated damping models are needed.

The most important sources of damping in timber structures are [20]:

- **Structural (Slip) Damping:** The motion in connections between different structural elements leads to energy dissipation due to friction, yielding of

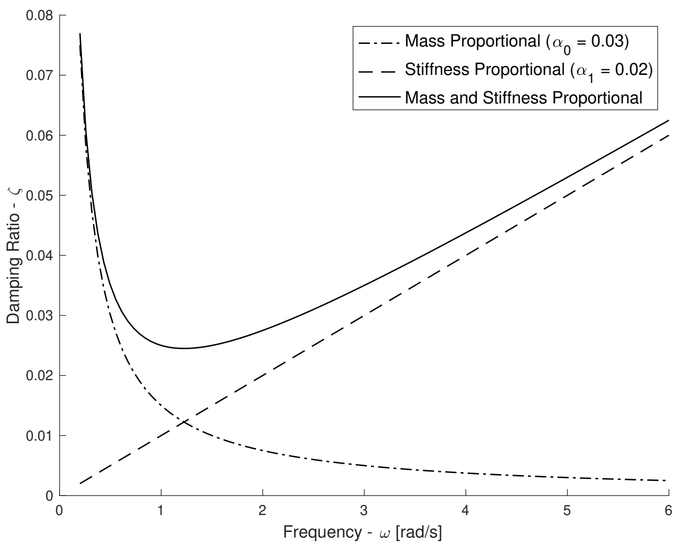


Figure 2.3: Mass and stiffness proportional damping ($\alpha = 0.03, \beta = 0.02$)

connectors and so on. Yeh et al. [20] found that the slip damping is up to 6-13 times greater than the internal damping, depending on the type of connection. Structural damping is usually taken to be proportional to the displacement or the force in the member, as opposed to viscous damping who is proportional to the velocity.

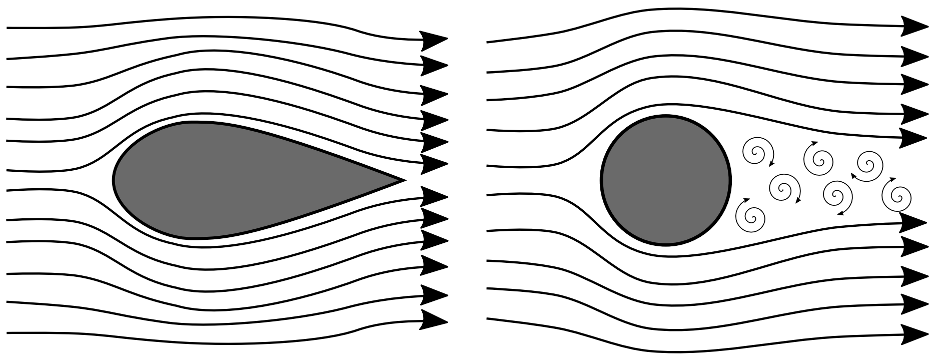
- **Material (Internal) Damping:** Material damping is a result of internal effects in the material, mainly internal friction.
- **Adhesive Damping:** Certain adhesive layers in a glued construction provide damping. According to [20] The adhesive damping is usually approx. 2 times greater than the material damping.

2.3 Wind Loads

Too large accelerations due to wind is a common cause of discomfort for occupants in the upper floors of a tall and slender building. Minimizing wind induced motion is therefore an important serviceability issue when designing tall buildings. The aim of this section is to cover some of the basics behind the highly complicated field of wind engineering.

2.3.1 Aerodynamics

Aerodynamics is the study of how air/gases interacts with objects (in this case buildings). There are two types of aerodynamic forces, lift and drag. Drag is the force acting in the wind direction, while lift acts perpendicular to the wind direction, i.e. vertically for a bridge or aircraft wing and horizontally for a building. The total drag on a body is the sum of "pressure" drag and "friction/viscous" drag. Pressure drag is caused by the drop in pressure behind a body, while friction drag is caused by the fluid (air) sticking to the body.



(a) Flow around a streamlined body **(b)** Flow around a bluff body

Figure 2.4: Air flow around different objects

Figure 2.4 shows the flow of air around two different types of cross sections, a "streamlined" body and a "bluff" body. It can be seen in Figure 2.4a that for a streamlined body the flow follows along the cross sectional shape, and that separation only occurs at the trailing end of the profile. Due to this, the main portion of the drag acting on a streamlined body is caused by friction, and less by pressure. However, a bluff body (Figure 2.4b) causes the flow to separate at some point before the trailing edge, leading to a relatively large "wake" region behind the object. The wake region causes the pressure behind the object to drop, as a consequence a bluff body experiences much higher pressure drag, but less drag caused by friction than a streamlined body [21]. Since virtually all civil engineering structures, including buildings and bridges are bluff bodies, the rest of this section will focus primarily on the excitation of bluff objects.

2.3.2 Buffeting Theory

The response of a building in the direction of the wind (along-wind excitation) is mainly caused by pressure drag. The response in the direction perpendicular to the wind (cross-wind direction) on the other hand, is more complex and is influenced by factors such as the building shape, turbulence and the shape and size of the wake [21].

The part of the load caused by variations in the wind velocity is the buffeting load. Buffeting load theory for bridges are presented by Strømmen [22], however the theory for towers are similar apart from some small changes in notation and other minor changes. The outline of the theory (for towers) are presented below.

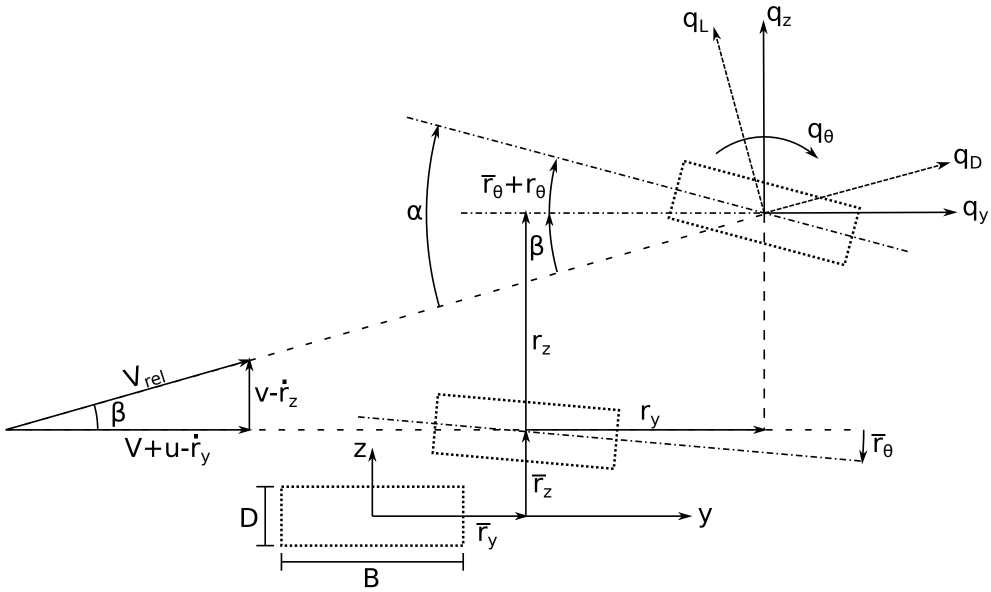


Figure 2.5: Flow and displacements (Modified version of fig. 5.1 in [22])

First a Cartesian coordinate system is established, where the x is the height coordinate, y is the coordinate in the along-wind direction and z is the cross-wind coordinate. It is assumed that the total wind velocity $U(x,t)$ is sampled over a limited period of time such that it can be split into a constant part $V(x)$ and a fluctuating part with zero mean $u(x,t)$ in the along-wind direction, in addition to fluctuating parts $v(x,t)$ and $w(x,t)$ in the horizontal and vertical cross-wind direction respectively. Figure 2.5 shows the cross section of a tower with dimensions $D \times B$, first the cross section is given a static displacement $(\overline{r}_y, \overline{r}_z, \overline{r}_\theta)$ by the time-invariant (mean) part of the wind action, this is the initial position of the vibrations caused by the fluctuating parts of the wind. The additional dynamic deformations caused by the fluctuating wind are denoted r_y, r_z and r_θ. In the axes of the wind flow coordinate system the drag, lift and moment acting on the cross section, in the deformed position, are given by the following matrix equation:

$$\begin{bmatrix} q_D(x,t) \\ q_L(x,t) \\ q_M(x,t) \end{bmatrix} = \frac{1}{2}\rho V_{rel}^2 \cdot \begin{bmatrix} D \cdot C_D(\alpha) \\ B \cdot C_L(\alpha) \\ B^2 \cdot C_M(\alpha) \end{bmatrix} \tag{2.16}$$

From Figure 2.5 it can be seen that the forces given in Equation 2.16 can be transformed to the global coordinate system using a transformation matrix who is a function of the angle β:

$$\beta = \arctan\left(\frac{v-\dot{r}_z}{V+u-\dot{r}_y}\right) \tag{2.17}$$

$$\begin{bmatrix} q_y(x,t) \\ q_z(x,t) \\ q_\theta(x,t) \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_D(x,t) \\ q_L(x,t) \\ q_M(x,t) \end{bmatrix} \tag{2.18}$$

An important assumption in buffeting load theory is that the fluctuating components of the wind velocity are much smaller than the constant component, hence $\beta \approx \frac{v-\dot{r}_z}{V}$, $\cos\beta \approx 1$ and $\sin\beta \approx \beta$. Then the wind actions in the global coordinate system become:

$$\begin{bmatrix} q_y \\ q_z \\ q_\theta \end{bmatrix} = \begin{bmatrix} 1 & -\beta & 0 \\ \beta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_D \\ q_L \\ q_M \end{bmatrix} = \begin{bmatrix} q_D - \beta\cdot q_L \\ \beta\cdot q_D + q_L \\ q_M \end{bmatrix} = \begin{bmatrix} q_D \\ q_L \\ q_M \end{bmatrix} + \beta\cdot\begin{bmatrix} -q_L \\ q_D \\ 0 \end{bmatrix} \tag{2.19}$$

The same assumption leads to:

$$V_{rel}^2 = (V+u-\dot{r}_y)^2 + (v-\dot{r}_z)^2 \approx V^2 + 2Vu - 2V\dot{r}_y \tag{2.20}$$

and

$$\alpha = \overline{r}_\theta + r_\theta + \frac{v}{V} - \frac{\dot{r}_z}{V} \tag{2.21}$$

It is also assumed that the force coefficients C_D, C_L and C_M can be approximated linearly:

$$C_D(\alpha) \approx C_D(\overline{\alpha}) + \alpha_f C_D'(\alpha_f) = \overline{C}_D + \alpha_f C_D' \tag{2.22a}$$

$$C_L(\alpha) \approx C_L(\overline{\alpha}) + \alpha_f C_L'(\alpha_f) = \overline{C}_L + \alpha_f C_L' \tag{2.22b}$$

$$C_M(\alpha) \approx C_M(\overline{\alpha}) + \alpha_f C_M'(\alpha_f) = \overline{C}_M + \alpha_f C_M' \tag{2.22c}$$

where:

$\overline{\alpha}$ = The angle caused by the mean velocity
α_f = The angle caused by the fluctuating velocity

Combining Equation 2.16, 2.19 and 2.22 gives the following expression:

$$\begin{bmatrix} q_y \\ q_z \\ q_\theta \end{bmatrix} = \frac{1}{2}\rho V_{rel}^2 \left(\begin{bmatrix} D\overline{C}_D \\ B\overline{C}_L \\ B^2\overline{C}_M \end{bmatrix} + \alpha_f \begin{bmatrix} DC_D' \\ BC_L' \\ B^2C_M' \end{bmatrix} + \beta \begin{bmatrix} -B\overline{C}_L \\ D\overline{C}_D \\ 0 \end{bmatrix} + \alpha_f\beta \begin{bmatrix} -BC_L' \\ DC_D' \\ 0 \end{bmatrix} \right) \tag{2.23}$$

As mentioned above both β and α_f are small, hence $\beta \cdot \alpha_f \approx 0$ i.e. the last term of the previous equation are negligible. Inserting the expressions for V_{rel} and α:

$$\begin{bmatrix} q_y \\ q_z \\ q_\theta \end{bmatrix} = \frac{1}{2}\rho(V^2 + 2Vu - 2V\dot{r}_y)\left(\begin{bmatrix} D\overline{C}_D \\ B\overline{C}_L \\ B^2\overline{C}_M \end{bmatrix} + (r_\theta + \frac{v}{V} - \frac{\dot{r}_z}{V})\begin{bmatrix} DC'_D \\ BC'_L \\ B^2C'_M \end{bmatrix} + \frac{v-\dot{r}_z}{V}\begin{bmatrix} -B\overline{C}_L \\ D\overline{C}_D \\ 0 \end{bmatrix}\right) \tag{2.24}$$

The wind action can be rewritten in terms of the mean wind load $\boldsymbol{q}$, the dynamic load caused by turbulence $\boldsymbol{B_q v}$ and the aerodynamic damping and stiffness matrices, $\boldsymbol{C_{ae}}$ and $\boldsymbol{K_{ae}}$:

$$\boldsymbol{q_{tot}} = \overline{\boldsymbol{q}} + \boldsymbol{B_q v} + \boldsymbol{C_{ae}}\dot{\boldsymbol{r}} + \boldsymbol{K_{ae} r} \tag{2.25}$$

where:

$$\boldsymbol{q_{tot}} = \begin{bmatrix} q_y & q_z & q_\theta \end{bmatrix}^T \tag{2.26a}$$

$$\overline{\boldsymbol{q}} = \frac{\rho V^2 B}{2}\begin{bmatrix} \frac{D}{B}\overline{C}_D \\ \overline{C}_L \\ B\overline{C}_M \end{bmatrix} \tag{2.26b}$$

$$\boldsymbol{B_q v} = \frac{\rho V}{2}\begin{bmatrix} 2D\overline{C}_D & DC'_D - B\overline{C}_L \\ 2B\overline{C}_L & BC'_L - D\overline{C}_D \\ 2B^2\overline{C}_M & B^2C'_M \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} \tag{2.26c}$$

$$\boldsymbol{C_{ae}}\dot{\boldsymbol{r}} = \frac{-\rho VB}{2}\begin{bmatrix} 2\frac{D}{B}\overline{C}_D & \frac{D}{B}C'_D - \overline{C}_L & 0 \\ 2\overline{C}_L & C'_L + \frac{D}{B}\overline{C}_D & 0 \\ 2B\overline{C}_M & BC'_M & 0 \end{bmatrix}\begin{bmatrix} \dot{r}_y \\ \dot{r}_z \\ \dot{r}_\theta \end{bmatrix} \tag{2.26d}$$

$$\boldsymbol{K_{ae} r} = \frac{\rho V^2 B}{2}\begin{bmatrix} 0 & 0 & \frac{D}{B}C'_D \\ 0 & 0 & C'_L \\ 0 & 0 & C'_M \end{bmatrix}\begin{bmatrix} r_y \\ r_z \\ r_\theta \end{bmatrix} \tag{2.26e}$$

The aerodynamic damping and stiffness matrices can be generalized using the mode shapes of the system, just like the structural matrices. If the load is deterministic (i.e the exact time history of the wind is known), the solution can be obtained in the time domain. However, in most cases the load is stochastic (i.e only the statistical properties like the mean and variation are known) and the solution is obtained in the frequency domain. The frequency response matrix which relates the load to the response in the frequency domain is:

$$\boldsymbol{H}(\omega) = [-\omega^2\tilde{\boldsymbol{M}} + i\omega(\tilde{\boldsymbol{C}} - \tilde{\boldsymbol{C}}_{\boldsymbol{ae}}) + (\tilde{\boldsymbol{K}} - \tilde{\boldsymbol{K}}_{\boldsymbol{ae}})]^{-1} \tag{2.27}$$

2.3.3 Eurocode

Eurocode 1 part 1-4 [23] provides rules for determining wind loads on civil engineering structures including buildings lower than 200 m and bridges with spans shorter than 200 m. The Eurocode uses a equivalent static load for determining the deformation caused by wind, while the appendix gives the formulas necessary to calculate the accelerations. As a consequence no dynamic analyzes, neither in the time or frequency domain, are needed when using the Eurocode for calculating wind response on a normal structure. The rules and recommendations are based on, among other things, the theory presented in the previous sections.

The first step in finding the static load is determining the basic wind velocity v_b:

$$v_b = c_{dir} \cdot c_{season} \cdot c_{alt} \cdot c_{prob} \cdot v_{b,0} \tag{2.28}$$

where:

c_{dir} = Directional factor (usually = 1.0)
c_{season} = Seasonal factor (usually = 1.0)
c_{alt} = Altitude factor (usually = 1.0)
c_{prob} = Probability factor (discussed in section "Return Period")
$v_{b,0}$ = Fundamental value of the basic wind velocity

The wind velocity and pressure consists of two parts, a mean value $v_m(z)$ and a fluctuating part described by the turbulence intensity $I_v(z)$.

$$v_m = c_r \cdot c_0 \cdot v_b \tag{2.29}$$

$$I_v = \frac{\sigma_v}{v_m} = \frac{k_l}{c_0 \cdot \ln(z/z_0)} \tag{2.30}$$

where:

$c_r(z)$ = Roughness coefficient ($= k_r \cdot \ln(\frac{z}{z_0})$)
c_0 = Orography factor (Usually = 1.0)
$\sigma_v(z)$ = Standard deviation of the turbulence ($= k_r \cdot v_b \cdot k_l$)
k_l = Turbulence factor (usually = 1.0)
z_0 = Roughness length

The next step is to calculate the peak velocity pressure. Note that the expression given here is from the national annex, but when $k_p = 3.5$ it becomes identical to the expression from the main part of the Eurocode.

$$q_p(z) = \frac{1}{2} \cdot \rho \cdot v_m^2(z) \cdot [1 + 2k_p I_v(z)] \tag{2.31}$$

where:

ρ = Air density (usually = 1.25)
k_p = Peak factor (= 3.5)

The wind pressure acting on the external surfaces are obtained from Equation 2.32. The internal pressure is often assumed to be of equal magnitude (see Figure 2.6) and opposite direction, i.e. it does not need to be taken into account when studying the building as a whole. The friction forces may be disregarded when the area of surfaces parallel to the wind is less than or equal to 4 times the area of the surfaces perpendicular to the wind direction. Hence for a global analysis of most tower-like structures it is only necessary to consider the external wind pressure.

$$w_e = q_p(z_e) \cdot c_{pe} \tag{2.32}$$

where:

z_e = The reference height of the surface/part
c_{pe} = External pressure coefficient

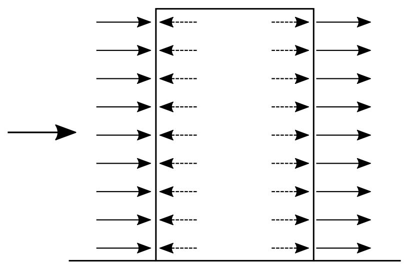


Figure 2.6: External (continuous arrows) and internal (dashed arrows) pressure

The total external wind force acting on the structure can then be found by summation of the product of the pressure from Equation 2.32 and the area of each part:

$$F_{w,e} = c_s c_d \cdot \sum_{surfaces} (w_e \cdot A_{ref}) \tag{2.33}$$

Alternatively, if $h/d > 5$, c_{pe} is not defined and the force is calculated directly using a force coefficient, c_f, given in the Eurocode:

$$F_{w,e} = c_s c_d \cdot \sum_{elements} (c_f \cdot q_p(z_e) \cdot A_{ref}) \tag{2.34}$$

where:

c_s = Size factor (discussed in section "Structural Factor")
c_d = Dynamic factor (discussed in section "Structural Factor")
c_f = Force coefficient of the structure
h = The height of the structure
d = The horizontal dimension of the structure in the wind direction
A_{ref} = The reference area of the respective surface

Structural Factor

The structural factor, $c_s c_d$ is the product of a size factor, c_s, and a dynamic factor, c_d. The size factor takes in to account that the peak pressure does not occur at the same time at all points in space of a large surface, the size factor cause a reduction in the load. The dynamic factor on the other hand, typically increases the load due to the effect of vibrations due to turbulence in resonance with the structure. For certain buildings with low slenderness $c_s c_d = 1.0$ can be used, however for tall, slender buildings c_s and c_d must be calculated using section 6.3.1 and annex B or C in the Eurocode.

$$c_s = \frac{1 + 7 \cdot I_v(z_s) \cdot \sqrt{B^2}}{1 + 7 \cdot I_v(z_s)} \tag{2.35}$$

$$c_d = \frac{1 + 2 \cdot k_p \cdot I_v(z_s) \cdot \sqrt{B^2 + R^2}}{1 + 7 \cdot I_v(z_s) \cdot \sqrt{B^2}} \tag{2.36}$$

where:

I_v = Turbulence intensity
z_s = Ref. height for structural factor (usually $= 0.6h$)
k_p = Peak factor (Note: not the same as in (2.31))
B^2 = Background factor
R^2 = Resonance response factor

Two different methods for calculating k_p, B^2 and R^2 are given in annex B and C, this text is based on the method in annex B. Only a selection of the expressions needed are presented below, the remaining expressions can be found in annex B of Eurocode 1991 part 1-4 [23].

The background factor B^2 takes in to account the lack of correlation of pressure on the surface. B^2 may be calculated using Equation 2.37, alternatively $B^2 = 1.0$ may be used as a conservative approach.

$$B^2 = \frac{1}{1 + 0.9 \cdot \left(\frac{b+h}{L(z_s)}\right)^{0.63}} \tag{2.37}$$

where:

b = Structure width
h = Structure height
$L(z_s)$ = Turbulent length scale

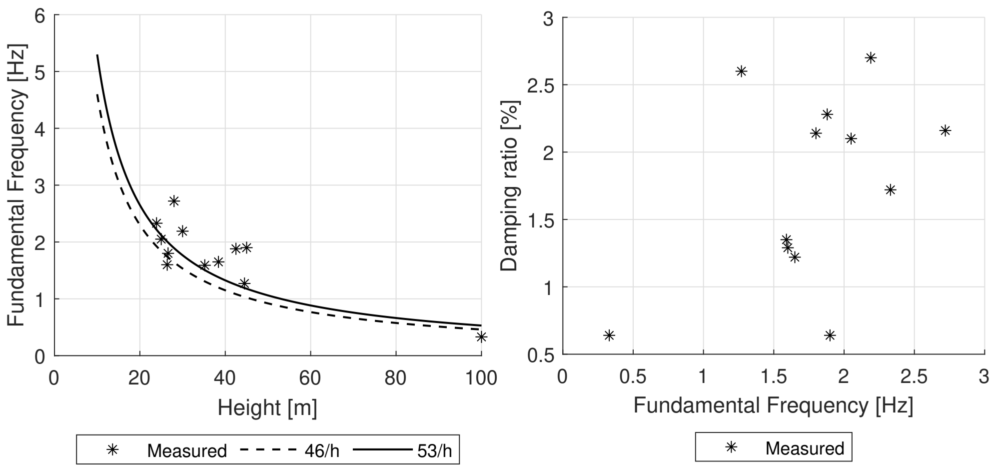
The peak factor, k_p, is the ratio between standard deviation and the peak value of the fluctuating component of the wind velocity.

$$k_p = \max\left(\sqrt{2 \cdot \ln(\nu \cdot T)} + \frac{0.6}{\sqrt{2 \cdot \ln(\nu \cdot T)}}; 0.6\right) \tag{2.38}$$

where:

ν = Upcrossing frequency
T = Averaging time for mean velocity ($= 600s$)

The upcrossing frequency is a function of the first natural frequency of the building, $n_{1,x}$. The Eurocode provides a simple estimate $n_{1,x} = 46/h$ to this frequency. However this estimate should be used with care, especially when dealing with timber structures, since the estimate is based on experiments performed on mainly concrete and steel towers. Feldman et. al. [24] performed a series of tests on a total of 12 timber structures (see Figure 2.7a) and found that $n_{1,x} = 53/h$ is a better approximation, but it should be noted that most of the structures tested were lower than what the Eurocode estimate originally is intended for. A way to lower the uncertainty drastically is to perform a modal analysis in a finite element program.



(a) First natural frequency vs. height

(b) Damping ratio vs. first natural frequency

Figure 2.7: Results from Feldman et. al. [24]

The resonance factor R^2 takes into account the possibility that the turbulence might be in resonance with the structure, a phenomena that can lead to a considerably higher load on the structure.

$$R^2 = \frac{\pi^2}{2 \cdot \delta} \cdot S_L(z_s, n_{1,x}) \cdot R_h(\eta_h) \cdot R_b(\eta_b) \tag{2.39}$$

where:

δ = Logarithmic decrement (damping)
S_L = Non-dimensional power spectral density function
R_h/R_b = Aerodynamic admittance functions

The Eurocode provides rough estimates of the logarithmic decrement, but again the source data for tall timber buildings are limited. Therefore the value of the logarithmic decrement are to a large extent based on the designer more or less guessing. In the study by Feldman et. al. [24] metioned above, it was also conducted tests of the damping ratio. A weak trend of the damping ratio increasing with the frequency can be seen in Figure 2.7b, however many outliers and a relatively small sample size makes it impossible to draw conclusions. One of the goals of this thesis is to find a way to be able to determine the damping more accurately.

Acceleration Response

The Eurocode also provides a simple method for estimating the acceleration response of a structure due to wind.

First the standard deviation of the acceleration, $\sigma_{a,x}(z)$, at height z is determined:

$$\sigma_{a,x}(z) = \frac{c_f \cdot \rho \cdot b \cdot I_v(z_s) \cdot v_m^2(z_s)}{m_{1,x}} \cdot R \cdot K_x \cdot \Phi_{1,x}(z) \tag{2.40}$$

where:

R = The square root of the resonant response R^2
K_x = Non-dimensional coefficient
$m_{1,x}$ = The along wind fundamental equivalent mass
$\Phi_{1,x}(z)$ = The fundamental along wind shape

The non-dimensional coefficient may be determined by Equation 2.41.

$$K_x = \frac{\int_0^h v_m^2(z)\Phi_{1,x}^2(z)dz}{v_m^2(z) \cdot \int_0^h \Phi_{1,x}^2(z)dz} \tag{2.41}$$

The along wind fundamental equivalent mass $m_{1,x}$ of a cantilevered structure can be approximated as the average mass per unit length of the upper third of the building.

Finally the peak acceleration at height z can be determined by multiplying the standard deviation (Equation 2.40) by the peak factor, k_p, defined by Equation 2.38 using the fundamental frequency, $n_{1,x}$, as the upcrossing frequency ν:

$$\ddot{u}_{peak}(z) = k_p(\nu = n_{1,x}) \cdot \sigma_{a,x}(z) \tag{2.42}$$

Return Period

Usually the return period, R, of the wind velocity are set to 50 years, i.e. the probability of exceeding that velocity in any given year, p, is 2%. However, it is sometimes necessary to calculate the actions for a different return period, e.g. when verifying the serviceability against the recommendations of ISO:10137 [25] a return period of only 1 year are used. The wind velocity is adjusted according to the return period using a probability factor, C_{prob}, in the equation for the basic wind velocity (Equation 2.28). The value of the probability factor is 1.0 when the return period are 50 years, and otherwise defined using the following formula:

$$C_{prob} = \left(\frac{1 - 0.2 \cdot \ln(-\ln(1-p))}{1 - 0.2 \cdot \ln(-\ln(0.98))} \right)^{0.5} \tag{2.43}$$

Where p is the annual exceedance probability. The usual method to calculate p is using Equation 2.44a, however this equation makes Equation 2.43 for C_{prob} undefined/invalid when the return period is short ($R \leq 1$). This issue can be solved by using Equation 2.44b to calculate p, as done by Talja and Fülüp [26] among others.

$$p = 1/R \tag{2.44a}$$

$$p = 1 - e^{-1/R} \tag{2.44b}$$

In Figure 2.8 the value of C_{prob} is plotted using both expressions for p, for return periods ranging from 1 to 100 years. It is evident that the values are almost identical, except that the exponential expression works better for short return periods.

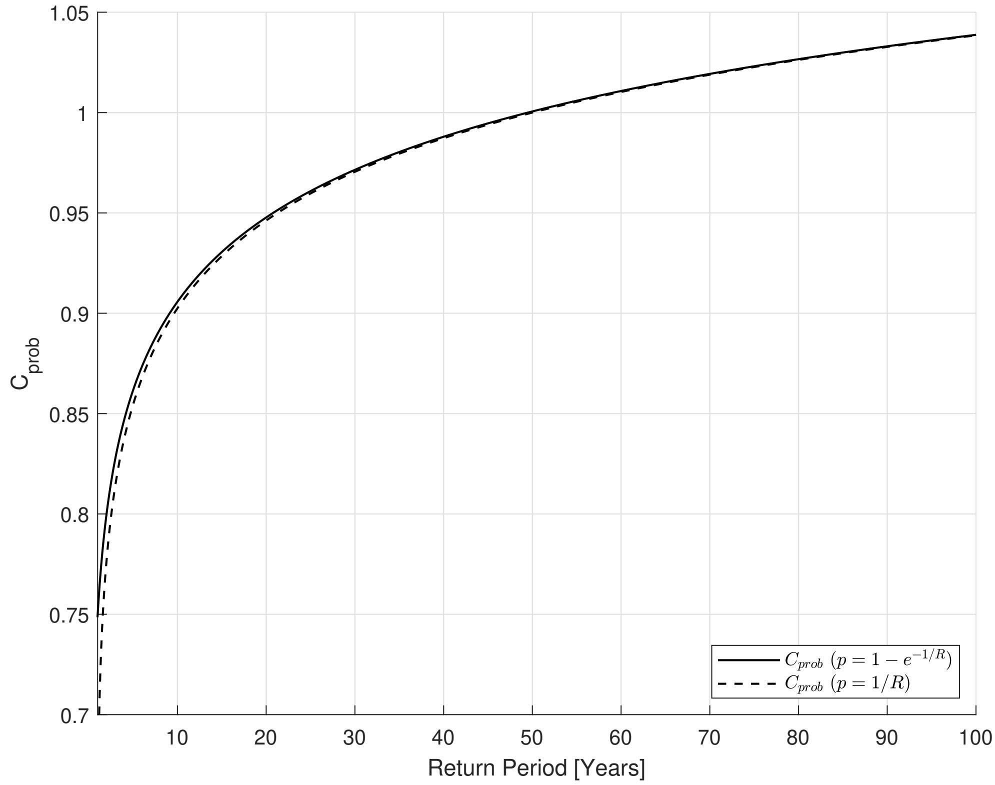


Figure 2.8: Probability factor C_{prob}

2.4 Finite Element Analysis

Finite Element Analysis (FEA) a widely used numerical analysis method. It is used for solving problems in many engineering disciplines, including structural engineering. The basic idea of FEA is that a system of complex behaviour can be divided into a finite number of non-overlapping subregions, also called elements. The behaviour of each of the elements can be described in a simple way. The elements are connected at certain points, called nodes, by requiring kinematic compatibility and static equilibrium at all nodes [27]. The result is a system where all components have a known behaviour, and thus by specifying how the components should interact with each other, the behaviour of the entire system can be determined. The accuracy of a finite element analysis depends on how many elements are used and the polynomial order of the interpolation functions. Generally, the higher the number of elements used is, the more accurate the solution will be. However, a large number of elements increases the number of equations that needs to be solved and therefore increases the computational time. A good finite element model balances adequate accuracy, while still keeping the computational time reasonably low.

2.4.1 Element Types

Numerous kinds of elements suitable for different types of problems have been developed. It is important to review the choice of element to use in a finite element analysis, as different elements have different capabilities, and not all are able to produce the wanted results for certain problems. The choice of element type will also greatly affect the computational time. The best approach is to choose the an element type that is as little computational expensive as possible, while still maintaining adequate accuracy of the model. This section will briefly explain the key points of some of the most used element types. The elements described are illustrated in Figure 2.9.

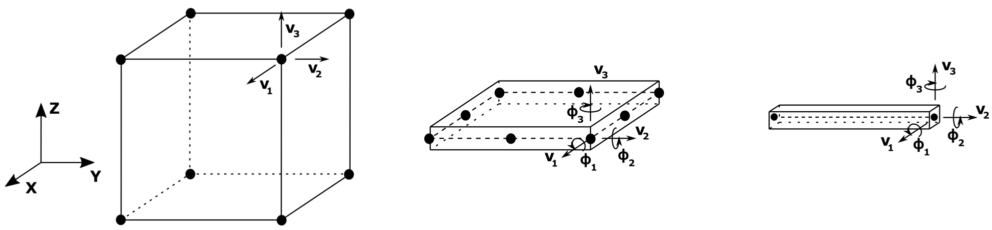


Figure 2.9: From left to right: Solid Element, Shell Element and Beam Element

Solid elements are the most general elements, and other elements, such as the beam and shell elements can be considered special cases of the solid element. The nodes of a solid element have all three translational degrees of freedom, and can therefore deform in three dimensions. Although the solid element can be used to model all kinds of structural components, it is also the element type that is the most demanding in terms of computational time. It should therefore only be used when it is required.

Shell elements can be used to model structures, where one of the dimensions is significantly smaller than the other two [28]. The body is discretized by defining the geometry only at a reference plane, thus reducing the number of nodes compared to solid elements, and then again the computational cost. The shell elements have both translational and rotational degrees of freedom.

Beam elements are one of the simplest kinds of elements, where both rotational and translational degrees of freedom are included. Beam elements are the one-dimensional approximation of a three-dimensional continuum [28]. The approximation is applicable for slender structures, that is, structures where the cross sectional dimensions are small compared to the length.

2.4.2 Beam Theory

Two main kinds of beam theory is used for structural engineering problems. The most widely used and simplest approach is the Euler-Bernoulli beam theory. This theory assumes that cross-sections that are plane and initially normal to the beam axis, remain plane and normal to the beam axis after deformation [29]. In other words, transverse shear deformations are not considered in this theory. The assumption is adequate for slender beams. For beams of uniform material, the dimensions of the cross-section should be less than 1/15 of the axial dimension of the beam, in order for for transverse shear flexibility to be negligible [28].

The other beam theory commonly used is the Timoshenko beam theory. Timoshenko theory includes transverse shear deformations, and is therefore the best choice for beams that are thicker and/or where the shear stiffness is low. In such beams the transverse shear flexibility can no longer be neglected, as doing so would result in a overly stiff beam. For a beam made of uniform material, Timoshenko beam elements are suitable for beams where the cross-sectional dimensions are up to 1/8 of the axial dimension [28].

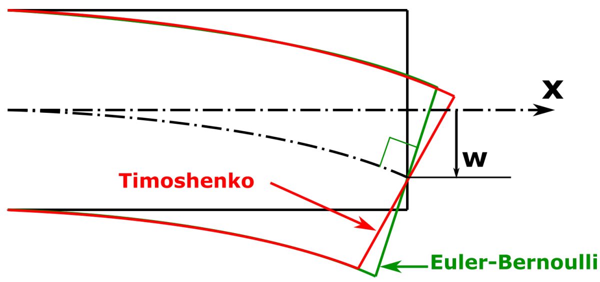
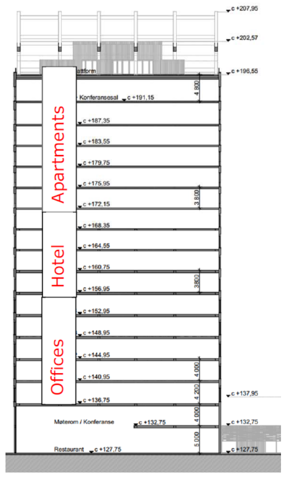


Figure 2.10: Euler-Bernoulli and Timoshenko beam theory

2.5 Mjøstårnet

Mjøstårnet is an 18-storey timber building with a height of 85.4 m completed in March 2019 and located in Brummundal in Norway [30]. At the time of writing it is the world's tallest timber building. Mjøstårnet is a multi-purpose building, housing offices, apartments and a hotel, see Figure 2.11b. The building is owned by AB Invest A/S, and the project was a collaboration between the contractor HENT, the architects Voll Arkitekter, the consulting engineering firm Sweco and the timber processing group Moelven. Several other companies have been involved with various subtasks, among them Woodcon/Stora Enso, who supplied the CLT used in staircases and balonies, and Ringsaker Vegg- og Takelement (RVT), who supplied the facade elements.

(a) Mjøstårnet during construction [30].



(b) Section of Mjøstårnet [31]. Note that the final elevations vary from what is shown here.

Figure 2.11: Mjøstårnet

2.5.1 Structural System and Materials

This subsection gives an overview of the structural system of Mjøstårnet. More detailed information can be found in [31]. The main load bearing system of Mjøstårnet is a glulam frame. The frame consists of beams and diagonals, as well as large scale glulam diagonals along the facades of the building. The beams and columns carries the global vertical forces, while the diagonals carry the horizontal forces applied to the building. Mjøstårnet has five shaft made of CLT panels: three elevator shafts and two staircases. The CLT panels carry the load from the stairs and elevators, but they are not designed to contribute to the horizontal stiffness of the building.

The base of the building is approximately 17 x 37 m^2. The foundation consists of a large concrete slab, supported by piles that are driven to bedrock. The piles can carry both compression and tensile forces.

There are to types of floors in Mjøstårnet. Floors 2 to 11 are consists of prefabricated timber elements, produced by Moelven. These are based on their Trä8 building system, and explained more in detail in subsection 4.2.1. Floors 12 to 18 are concrete floors. These are a combination of a prefabricated bottom part that is used as formwork for a cast in place upper part. The reason for using concrete in the upper floors is that they result in an increased mass at the top of the building. This is favorable as it results in larger inertia forces, and thus reduces the accelerations in the top floors. The concrete floors are also favorable for the acoustic performance. All floors are designed to act as diaphragms, and are supported by glulam beams in the frame.

A large pergola structure made of glulam is fixed to the roof, which will also be used as a terrace. In addition to this an apartment is placed on top of the roof. On the residential floors, floor 12 to 17, balconies are fixed to the side of the building. The balcony decks are made of CLT. The different structural components can be seen in Figure 2.12.

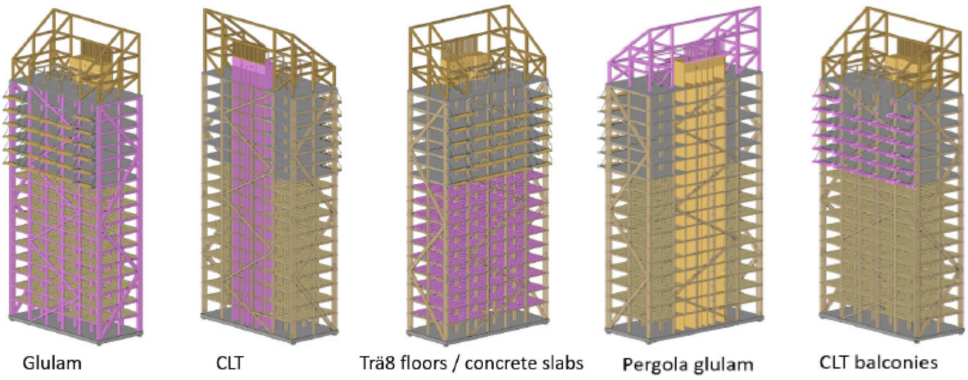


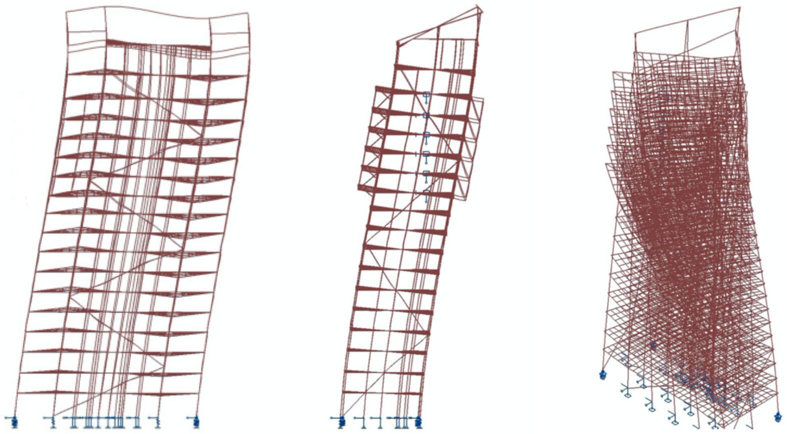
Figure 2.12: The structural components of Mjøstårnet [30]

The facade is made of prefabricated elements. The insulation, windows and external cladding are located within these elements. In the design process, the stiffness of the wall elements are not considered to contribute to the global stiffness of the building. By placing the wall elements outside the frame, climate class 1 can be used for all structural timber members except the pergola.

All of the glulam elements are connected by slotted-in steel plates and dowels. Strenght classes GL30c and GL30h according to EN 14080:2013 [32] are used for the glulam members in the building. The CLT used has a bending strength of f_{mk} = 24 MPa.

2.5.2 Numerical Model

During design, a numerical model of Mjøstårnet was developed by Sweco using the structural analysis software Autodesk Robot Structural Analysis 2017. The structural damping ratio used for the model was $\zeta = 1.9$ %. A modal analysis was conducted and Figure 2.13 shows the three first vibration modes. The corresponding frequencies are shown in Table 2.3a. Mode 1 and 2 are bending in the longitudinal and transverse directions respectively, while mode 3 is torsional.



(a) Mode 1: Longitudinal **(b)** Mode 2: Transversal **(c)** Mode 3: Torsional

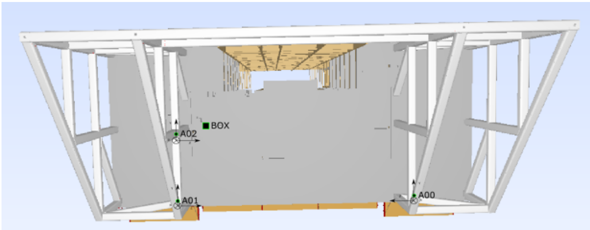
Figure 2.13: Mode shapes from numerical modal analysis [33]

2.5.3 Monitoring and Measurements

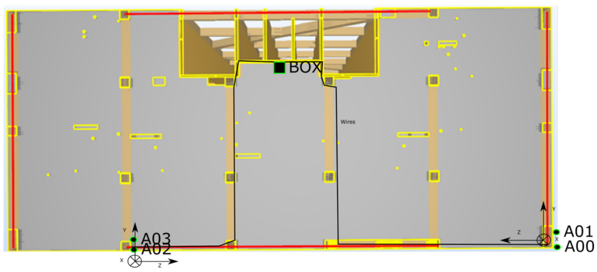
Due to the relatively small amount of tall timber buildings in the world, the amount of empirical data on this kind of structure is very limited. This lead to the structural behaviour being unpredictable, as a lot of design decisions have to be made based on assumptions and qualified guesses. In order to make timber a more attractive structural material, it is important to utilize the buildings that already are constructed in order to produce empirical data. The data can then be used in order to verify and support numerical models, and thus improve the predictions of how future structures will behave.

Monitoring equipment has been installed at Mjøstårnet in order to measure vibrations. The measurements are done by three accelerometer pairs, all fixed to the pergola (see Figure 2.14a). In addition to the sensors in the completed structure, a temporary set of accelerometers were installed during construction. These were placed on floor 7 at the positions shown in Figure 2.14b. In both cases, the sensors are only placed at one level. It is therefore not possible to obtain the exact mode shapes of the structures from these measurements [33].

Based on measurements of ambient vibrations due to wind loading, Tulebekova *et al.* [33] identified 8 stable modes by using the data-driven stochastic subspace identification technique (DD-SSI). Modes 1-3 are all below 1 Hz. This is within the expected range for a structure of this size, and modes 1-3 are therefore considered as whole structural modes. The frequencies of modes 1-3 are shown in table 2.3b. Modes 4-8 are in the range 1.8-4.7 Hz, and considered to be local modes of the pergola structure.



(a) After completion



(b) During construction

Figure 2.14: Setup of monitoring system [34]

Table 2.3: Fundamental frequencies of Mjøstårnet

(a) Numerical model by SWECO

Mode	**Frequency** [Hz]	**Mode directionality**
1	0.33	Longitudinal
2	0.37	Transverse
3	0.59	Torsional

(b) Measured (DD-SSI)

Mode	**Frequency** [Hz]	**Mode directionality**
1	0.50	Transverse
2	0.54	Longitudinal
3	0.82	Torsional

By comparing the results, it is clear that the numerical model produces frequencies that are lower for all three modes. It can be seen that the frequencies of mode 1 and 2 are close for both cases. In addition, the directionality of the first two modes are switched. Tulebekova *et al.* [33] suggests that the lower frequencies in the numerical model are due to underestimated foundation stiffness in the model. The change directionality might occure due to two factors: closely spaced modes and incorrect foundation stiffness in the model.

Chapter 3

Modelling

A large portion of the time spent working with this thesis has been dedicated to the development of a parametric model of a tall timber building. The parametric model is programmed in Python 2.7 [35] and is designed to run in Simulia's finite element analysis (FEA) software Abaqus [36]. The script has primarily been developed and tested in Abaqus 2019, but is likely to work in other versions as well. All of the user input is made in a Microsoft Excel workbook, hence little or no prior knowledge of Python is necessary for basic use of the model. This chapter describes modelling choices and assumptions made when creating the model. A user manual for setting up the input workbook and running the script is provided in Appendix A.

3.1 Choice of Software

Two deciding factors for choosing suitable software for this thesis was established at an early stage. The first factor being that the program need to be as general as possible and not put limitations on what parameters it is possible to study. The second factor is the fact that the program needed to allow for a parametric approach, making it simple to run analyses where different parameters can be easily adjusted. The FEA software Abaqus was deemed to be the best option, as it is a powerful and well documented general-purpose FEA program capable of running analyses of complex models and giving the user full control of the parameters of the model. This comes at a price, as modelling with Abaqus can be a more tedious and demanding process compared to modelling in FEA programs that are specialized on civil engineering structures. However, such programs is found to be insufficient for this thesis as their modelling options is likely be too limited. In addition to its modelling capabilities, Abaqus can be run through Python scripts, making it suitable for performing parametric studies. Finally, Abaqus is compatible with Simulia's analysis tool Isight [37]. Isight allows the user to automate simulations and by this greatly improves the efficiency of running a parametric study.

3.2 Model Overview and Limitations

Before getting into the specifics of the model, this section will present the assumptions and limitations made before the development of the model started.

Type of Building

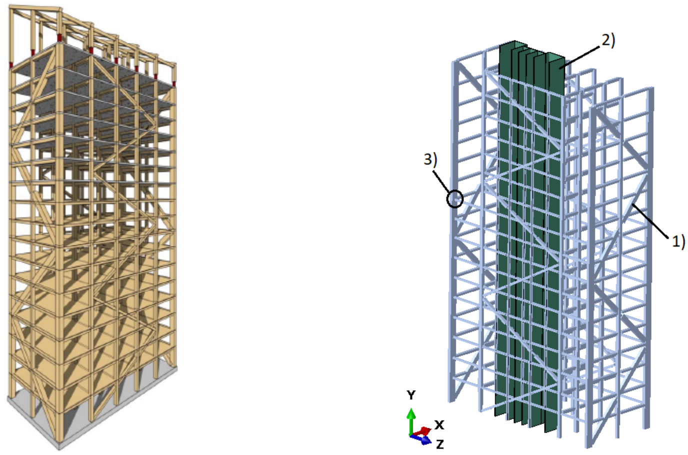
Due to the limited time available for the work of this thesis, creating a model that is capable of representing all kinds of tall timber buildings is not achievable. In addition, a completely generalized model would be just as easy to achieve by modelling directly in Abaqus, as the user input needed for such a script would be very comprehensive. One of the main benefits of a parametric model would then have been lost.

The model is limited to only cover buildings using a post and beam system as the main load carrying system. This system is characterised by a skeleton structure consisting of columns and beams, typically made of glulam. The post and beam system will hereafter be referred to as the frame. The reason for focusing on this system, is that Mjøstårnet, the case building that will be studied in the thesis, is built with this system. The entire script has been highly influenced by the structural system of Mjøstårnet in order to model the building as correctly as possible.

Horizontal stiffness can be added to the model by three different approaches: diagonal bracing, shear walls in the form of shafts and moment-stiff joints. The different kinds of bracing options are shown in Figure 3.1b. The script requires that the building uses diagonal bracing, while the two other approaches are optional.

Coordinate System

The horizontal plane of the model is defined as the XZ-plane. The X-direction is defining the direction that internal beams, used for supporting the floors, are



(a) The post and beam system is the main load carrying system of Mjøstårnet.

(b) Approaches for adding horizontal stiffness: 1) Diagonal bracing, 2) Shafts, 3) Moment-stiff joints.

Figure 3.1: Structural system of the parametric model

allowed to span. These beams typically span along the short side of the building. From this point the X-direction will be referred to as the transverse direction of the building. The Z-direction is typically defined as the direction along the long side of the building, and is hereafter referred to as the longitudinal direction. The Y-direction is defining the vertical direction. The axis system is shown in Figure 3.1b.

Grid Reference System

The frame structure of Mjøstårnet is highly repetitive. A grid reference system is therefore implemented for defining the geometry. In the horizontal plane, the grid lines define the position of columns. Columns may only be placed at the grid line intersections, as seen in Figure 3.2. The user also has to specify the vertical coordinates indicating the positions of all levels of the building. This grid reference system can be used in order to place most structural members in the building. The system simplify the user input as the user only have to specify the location of the grid lines and levels once.

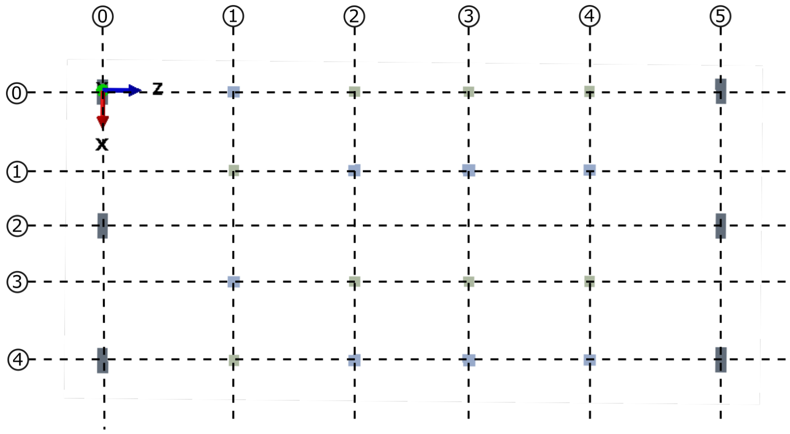


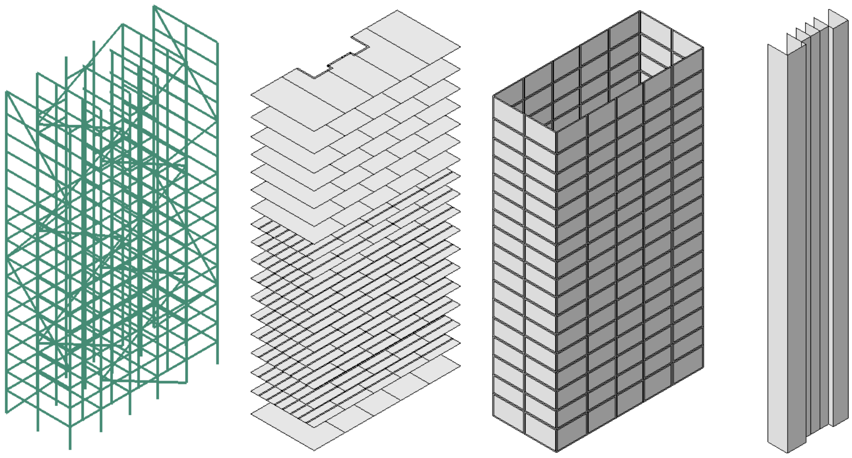
Figure 3.2: Grid lines and columns in the horizontal plane. Note that all columns are placed at grid line intersections, but not all intersections hosts a column.

Parts

Modelling in Abaqus requires the user to first define parts. Each part is defined in a local coordinate system and is independent of all other parts, before the parts are assembled in a global coordinate system in the assembly module. In addition to placing the parts relative to one another, the module is used to define how the parts should interact, such as applying connections between them.

The most obvious approach would be to model each member as an individual part, and then assemble the parts as a building in the assembly module with the use of the built in connection tools of Abaqus. However, another way of modelling the connections was chosen, this is explained in section 3.7. This approach does not require the model to have individual parts for each member, as the connection properties are assigned directly to the parts themselves. All members that overlap in a part, will automatically be tied. This can be used as an advantage as it removes the challenging process of defining interaction properties between many parts.

The chosen approach for modelling connections does in fact allow for the entire model to be defined as one part. However, this would prove difficult, as it would make assigning different properties to different members a challenging task. A middle ground approach is therefore chosen. The model is split into four individual parts. The first one being the frame part. This part host all beams, columns and diagonals. The second one is the floor part, which host all the different floor decks. The third part is the outer wall part, hereafter simply called the wall part. Finally, we have the shaft part which is hosting all shafts. The different parts will be discussed further on a later stage of this chapter. Each part consists of multiple members, thus reducing the number of constraints that needs to be added in the assembly module. At the same time the parts are small enough to make it possible to access and alter the properties of each member. The different parts are shown in figure 3.3.



(a) Frame Part **(b)** Floor Part **(c)** Wall Part **(d)** Shaft Part

Figure 3.3: The four different parts of the model

Finite Element Types

An important feature of a parametric model, is that it has to be relatively computational inexpensive. This is necessary since many simulations are required in order to conduct a parametric study. In order to keep the model as computational inexpensive as possible, while still maintaining adequate accuracy, the choice of elements is important. The entire frame part is therefore modeled using beam elements, while the floor, shaft and wall parts all are modelled using shell elements. Shell element sections are defined by a thickness and a material. Many of the components that will be modelled by shell elements are far more complex than this, often consisting of multiple materials and complex section geometry. In these cases, preliminary studies is required in order to find the shell section properties that match the properties of the real cross-section. An example of how this can be done is given in subsection 4.2.1.

3.3 Frame

As already stated, the script is able to model buildings which use a post and beam structural system as the main load carrying system. This, together with the diagonal bracing, is what constitutes the frame part of the model. To see how the connections within the frame are modeled, see section 3.7. All members of the frame part are modelled using beam elements.

3.3.1 Columns and Beams

The frame part is based around the geometry of a generic frame which is defined using the grid system. This generic geometry will hereafter be referred to as the base frame. Columns are placed at user specified intersections of the grid lines, and span from the first to the last level of the building. Beams are placed at every level, except from the first. In the longitudinal direction (z-direction), beams are only placed along the outer grid lines. However, in the transverse direction (x-direction) beams are placed at every grid line. Thus, the internal beams only span in the transverse direction. The placement of beams and columns in the horizontal plane can be seen in Figure 3.5. The resulting base frame is shown in Figure 3.4.

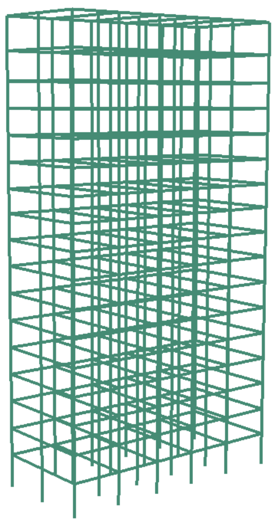
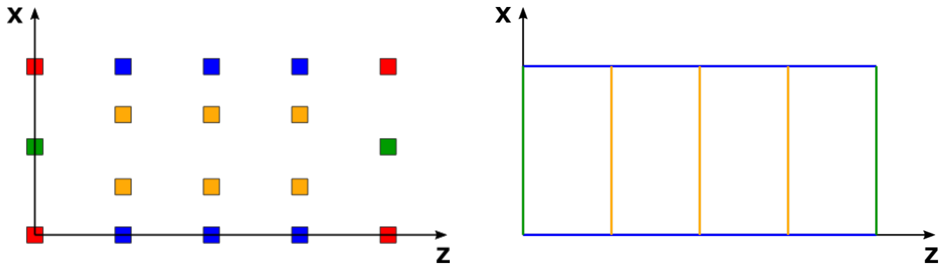


Figure 3.4: Base frame defined by grid

It was decided to make some restrictions when it comes to the amount of different cross-sections that can be used to the model. The simplification is done by defining sets or groups of the members, that are repeated across the entire height of the structure and assigned the same properties. The columns are separated into four groups: corner columns, long edge columns, short edge columns and inner columns. The beams are separated into three groups: long edge beams, short edge beams and inner beams. The different groups are shown in Figure 3.5. The reason for implementing these restrictions was to simplify the user input. Having to input the cross-section parameters of every member of the frame would be very time consuming and make it hard to keep track of all the input. In a real building, column dimensions may be reduced towards the top, and the cross-section of beams will vary in size depending on what type of floor they are supporting and how long the spans are. However, it was deemed that using average dimensions for the members within each group would be a satisfactory simplification for determining the dynamic behaviour of the total system.



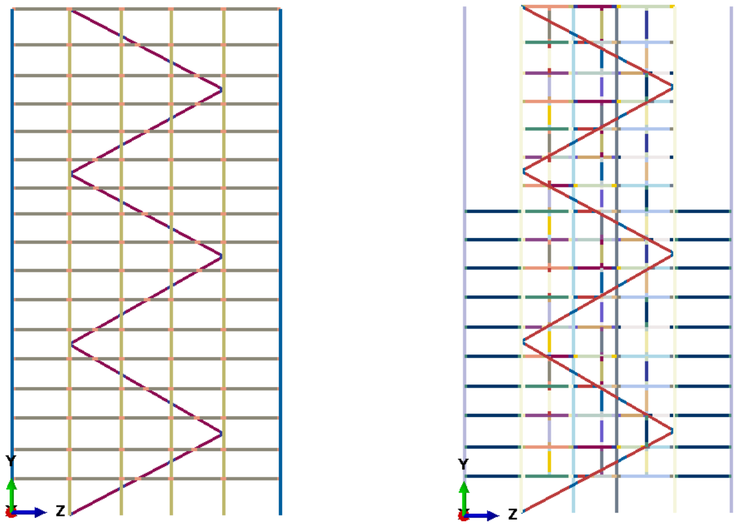
(a) Column groups. Red: Corner columns, Blue: Long edge columns, Green: Short edge columns, Yellow: Inner columns.

(b) Beam groups. Red: Long edge beams, Green: Short edge beams, Yellow: Inner beams.

Figure 3.5: Cross sectional groups for beams and columns, viewed in the horizontal plane.

The user is given the option to alter the generic geometry of the base frame. This can be done by removing any beam or column. It is possible to remove parts of a column, such that the resulting column does not span the entire height of the building. It is also possible to add single beams and columns to the frame. This added members must span in either x-, y- or z-direction, but the placement is not restricted to the grid, as the start and end points are defined by coordinates and not indices of the grid. Each of the added members must be assigned a material and cross-sectional properties. The user can decide if the members should include connector segments as explained in section 3.7. If connector segments are included, their properties must also be defined.

Diagonals are not affected when columns and beams are removed and added. This includes the connector segments of the diagonals, which may result in a few connector segments that are out of place. This has not been fixed due to limited time, but it is also assumed to be insignificant for the performance of the total system. The user is also able to remove the long edge beams from specified levels. If a diagonal intersects with a long edge beam in one of these levels, the beam in the span where the intersection takes place will not be removed. All the options available for altering the base frame geometry, allows for great flexibility. An example of how the base frame can be altered is shown in Figure 3.6.



(a) Base frame geometry

(b) Altered frame geometry.

Figure 3.6: Example of how the base frame can be altered. Each color represent an individually defined cross-section. Notice that the diagonal is not affected.

3.3.2 Diagonals

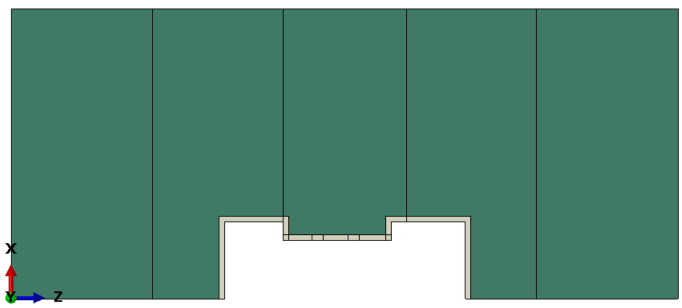
In addition to columns and beams, the frame part is hosting the diagonal bracing members. Diagonals are required in order for the script to run, and need to be placed in both the xy- and yz-plane. The diagonals are not required to be placed in the outer walls of the building, but can be placed at the grid lines desired. Diagonals are grouped based on if they are spanning in longitudinal or transverse direction. For each diagonal group, the user can define the start and end level, the start and end column, how many levels each diagonal should span across and the vertical placement of the turning points of the diagonals. In addition, cross-sectional parameters and material is defined for each group.

3.4 Floors

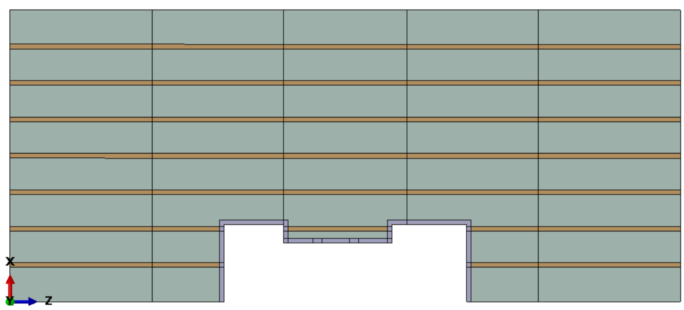
The floor part is hosting all floors, including the foundation slab. The floors are modelled as shell elements, and the parameters of the floors are therefore the shell thickness and material.

Timber buildings can have various types of floor. Different kinds of floor elements made of timber is one option, but there are also examples of timber buildings with concrete floors. Mjøstårnet utilizes both of these options. The concrete decks are a combination of prefabricated elements and cast in place concrete. In order to accommodate for various kinds of floors, an option for modelling the floors as element based floors or as continuous decks is included into the script. Continuous decks are simply modelled as large continuous shells without any variations in properties.

The module based floors, on the other hand, include so called "connection-zones". These zones are parts of the shell that can be given different properties in order to simulate the softer behaviour of connections between the modules. This is explained further in subsection 3.7.2. Both thickness and material can be assigned separately to the connection-zones. The difference between the two types of floors can be seen in Figure 3.7. Preliminary tests showed that if the connection-zones are assigned very low stiffness, local modes with frequencies interfering with the global modes may occur.



(a) Continuous deck



(b) Module based floor. The yellow areas represent the connection-zones, that is areas of the floors that can be assigned separate properties.

Figure 3.7: Floor types. Openings for the shafts are included in the floors. Note that the small areas surrounding the openings have different section properties, in order to simulate the connection between floor and shaft, see section 3.6.

The floors are connected to the transversal beams of the frame. From tests during development of the model, it was found that the stiffness of the floors was of little importance to the natural frequencies of the building. It was therefore decided not to include connection-zones representing the floor-to-frame connections. In other words, this means that the floors are tied to the frame without stiffness reduction. One disadvantage of not representing the floor-to-frame connections, is that it will not be possible to study the effect of adding damping in these connections directly.

If shafts are included in the model, the script will create openings in the floor around the shafts. If the shaft should be tied to the building a connection-zone is also placed around the shaft openings, in order to simulate connection between the floors and the shafts.

The floor placed at ground floor, the foundation slab, is of little importance to the model. The approach chosen for modelling the foundations assign the foundation stiffness directly to the columns, see section 3.8. The main purpose of the slab in the model is to create a tie to the lower parts of the walls, and thus prevent local modes to appear.

3.5 Walls

In a timber building with a frame as the main load carrying system, the outer walls are rarely designed to be carrying any loads other than its self weight. This was also the case when Mjøstårnet was designed. The numerical model made by Sweco during design of the building does not include the stiffness contribution that the wall elements would provide. Nonetheless, it is obvious that non-structural wall panels will affect the dynamic properties of a building. Although wall panels usually are considered non-structural elements, they will add some stiffness to the structure that they are fixed to. In addition, under dynamic response, friction between the wall panels and other members are likely to occur, thus adding structural damping. In order to study these effects, outer wall panels are implemented into the model.

The walls are modelled using shell elements, and their properties are defined by a shell thickness and material. The wall part is tied to the frame part, and the user is able to decide if they only should be connected to the beams (floors for levels without beams), or both to beams and columns.

As explained in subsection 2.5.1, the facade of Mjøstårnet consist of prefabricated elements. It is likely that the connections between the wall elements, and between the wall elements and the frame, are softer than the wall elements themselves. In order to simulate this, connection-zones are created at the edges of each wall panel. This is illustrated in 3.8. The walls are automatically partitioned based on the grid lines and levels. Although this automated partitioning does not allow for customization and fully accurate modelling of the walls, it was deemed to be sufficiently accurate for this type of facade.

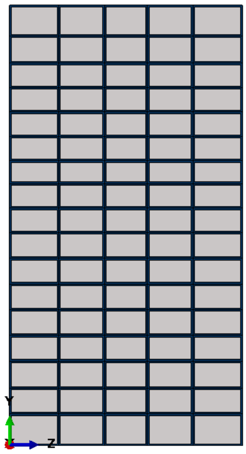


Figure 3.8: Partition of walls in order to simulate connections. The gray area is assigned with original cross section parameters, while the dark blue areas are assigned with connection properties.

3.6 Shafts

Every tall building have one or more shafts. The shafts are typically used for housing elevators or staircases. Technical installations such as ducts for the ventilation system, electrical wiring and plumbing are also commonly placed in shafts. Some structures utilize the shafts to provide lateral stiffness, however this is not the case for Mjøstårnet [31] and Treet [15]. For tall timber buildings using shafts to provide lateral stiffness, the shafts is often made in concrete, due to higher material stiffness. Since the structural design of Mjøstårnet (and Treet) does not require the higher stiffness of concrete, the shafts are built in cross laminated timber (CLT).

Including one or more shafts in the model is required in order for the script to run correctly. To accommodate for the many different types of shafts, various options are implemented in the model. The shafts are modeled using shell elements, and can be assigned any material and section thickness. The location are set using coordinates instead of axis numbers, so they can be placed anywhere inside the building and are not bound to the grid. The user can choose not to connect the shaft to the floors. In this case the the shaft itself will not be modelled, but there will be made holes in the floors. The user also has the option to remove one of the shaft walls.

The script only allows for a single material to be assigned to all shafts. For most buildings, this is an accurate simplification. Also, if the shaft is made of a laminate or composite such as CLT, the properties of the laminate needs to be modelled into the material, since the script currently only supports homogeneous shell sections. This can done by e.g defining a fictitious orthotropic material with parameters that causes a similar behavior as the original laminate. For a study of the global structural behavior this simplification is acceptable. It is also still possible to define more accurate composite cross section manually in the Abaqus GUI after the model is generated, if necessary.

3.7 Connections

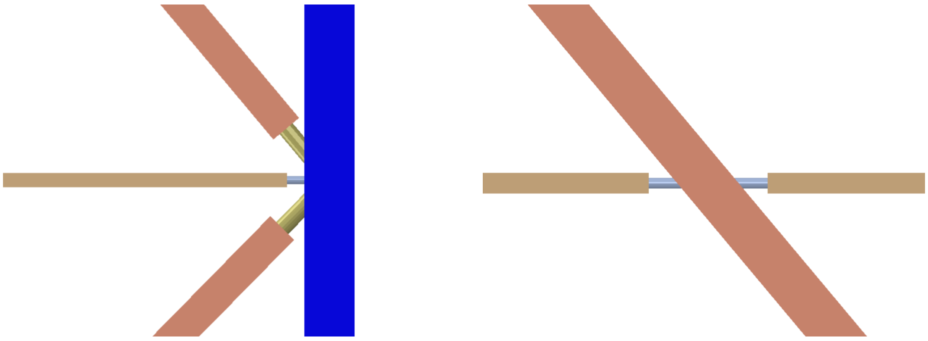
3.7.1 Connections of Beam-type Members

In structural analysis it is common to idealize joints either as pinned or rigid, meaning that no moment or all the moment are transferred from one member to another trough the connection. In reality however, all joints are semi-rigid, i.e. no connection is completely free to rotate nor completely stiff. The overall stiffness, and therefore also the dynamic properties, of a structure depends heavily on the stiffness of the connections, hence it is important to represent the connections as accurate as possible when modelling and analyzing the structure.

In the parametric model made as a part of this thesis, the connections between columns, beams and diagonals are originally modelled as rigid, but a with a short segment at the end of diagonal and beam. To account for the reduced axial and rotational stiffness in a connection the user can specify a fraction of the original area and/or second moment of area to be assigned to the connector segment. The connector segments are assigned a generalized cross section in Abaqus, meaning that area, second moment of area about both axes and the torsional constant can be defined independently of each other. Hence it is possible to create a connection that is e.g. stiff when loaded axially, but almost free to rotate, or vice versa. The mass density are adjusted automatically in the script such that the total mass is unchanged.

In connections between a column, a diagonal and a beam, the column retains its original cross section while the diagonal and beam gets a connector segment. Similarly, only the beam gets connector segments when connected to a diagonal or column (see Figure 3.9). The placement of the connector segments are selected to represent the actual connections on Mjøstårnet (Figure 3.10) as realistically as possible.

The columns are modelled as continuous along its entire length. This is because butt joints are assumed to retain most of the stiffness of the column when loaded in compression [38].



(a) Connection between a column, a beam and two diagonals

(b) Connection between a diagonal and two beams

Figure 3.9: Connections as modelled i Abaqus

(a) Connection between a column, a beam and two diagonals

(b) Connection between a diagonal and two beams

Figure 3.10: Connections as built, taken from IFC model provided by Sweco

Another approach to implement semi-rigid connection in the model is to create separate parts for the columns, beams and diagonals and connect them using springs/connector elements in Abaqus. However this method would be more complicated to implement, especially when dealing with 3D structures, and more prone to severe errors, such as singularities, causing the analysis to fail. Therefore the method of reducing the cross sections in parts of the member was deemed the best for the purpose of this thesis, even though using springs/connector elements is more correct in theory. Another benefit of the method chosen is that it is relatively easy to understand the input, who simply is fractions of the original cross section properties. Utne [39] employed the same principle of modelling the connections in her study of the dynamical properties of the tall timber building "Treet" in Bergen.

3.7.2 Connections of Shell-type Members

Walls, floor and shaft are modelled with shell elements, and it is therefore required have a solution for modelling connectors for shell-type members as well. The approach chosen is similar to what was chosen for the beam-type members. The shell part is in it self modelled as one continuous part, but is partitioned with so called connector-zones that can be assigned with separate properties. The placement of the connector-zones is done differently, depending on the part it belongs to. This is explained in the sections of the respective parts.

A few different options for altering the stiffness of the connection-zones have been implemented in the script. First of all, the user have to specify the width of the zone, which impact the rotational stiffness. Unfortunately, the method of using generalized section as done for the beam-type connections is not possible to use for shell-type sections. The two properties that define a shell section are the thickness and the material, and both can be defined by the user for the connection-zones. The thickness is specified as a fraction of the original thickness of the shell part. Very low thicknesses should be used with caution, as it may lead to unwanted local modes occurring in the connection-zones. Only being able to use the thickness is more restrictive compared to the generalized sections used for the beam-type connections. It is, for instance, not possible to alter the rotational and membrane stiffness separately. It is also hard to relate the connection stiffness directly to the stiffness of the original section. The option of choosing another material for the connection-zone was therefore implemented, as it gives more freedom in the modelling process.

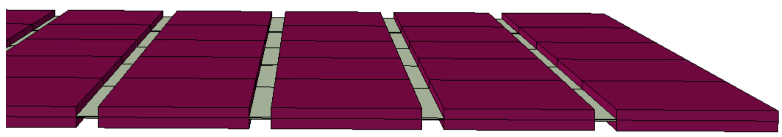


Figure 3.11: An example of how shell-type connections are modelled using connection-zones. The connection-zones are the light grey areas. Both the shell thickness and the material can be altered in the connection-zones.

Another approach for choosing the connector stiffness of shell parts was also considered. This was to use the "General Shell Stiffness" approach for defining the shell section that is available in *Abaqus*. The option gives full freedom, but it requires the user to assemble the stiffness matrix of the section, and was therefore considered to be overly intricate. In addition, the approach does not allow for damping to be represented in the connection.

As opposed to the beam-type connections, the density of the shell connection-zone material is not altered in order to compensate for the reduced thickness. This is something the user needs to consider, as it will alter the mass matrix of the structure. However, in most cases the change is likely to be negligible. Why this was done can be explained by the additional option of assigning a separate material to the connection zone. If the connection material have a density that is not equal to the original material of the member, the scaling of the density would be out of place.

3.8 Foundation

Even though a timber building is less heavy than its counterparts made of steel or concrete there is still large forces that needs to be transferred to the ground through the foundation, usually made of piles. The foundation stiffness are modelled with six springs to ground at the bottom of each column and shaft corner. Three springs are placed in the global x-, y- and z-direction respectively. In addition there are rotational springs about each axis. To be able to model damping, each degree of freedom are also equipped with dashpot dampers. The user gets the possibility to add a dashpot coefficient which relates the relative velocity to the damping force, individually for each DOF. Both the springs and dashpots defined for the foundation are linear, and the option in Abaqus to allow for e.g. temperature dependency was not deemed necessary to implement in the code for the parametric model.

3.9 Loads and Non-Structural Mass

Along with mass from the structural components that have been presented earlier is the possibility to add non-structural mass. Both distributed mass and point mass can be added to the structure. The distributed mass can be added to one or multiple floors, and is typically suitable for representing live loads. Point masses can be added to any point of the grid. Point masses can be used to include the mass of components that is expected to not add stiffness to the model, such as balconies.

3.10 Wind Load

There are two different ways of calculating and applying wind loads to the structure, either by using the method given in Eurocode 1 part 1-4 [23] or by specifying a time history of the loading.

- The method given in the Eurocode is based on the calculation of equivalent static load for determining the displacement due to the loads. The procedure is explained in detail in subsection 2.3.3 and demonstrated through a series of different tests in chapter 7. The wind forces calculated are converted to line loads and applied to the columns of the frame structure.
- The second method is to specify a time history of the load. The source from time history can either be on-site measurements or it can be generated from a spectrum. A procedure for generating a time series from a spectrum is not implemented in the code, but can be found in e.g. appendix A of Strømmen [22]. The pressure load is then applied as a pressure load acting on the surface of one of the exterior walls, as shown in Figure 3.12.

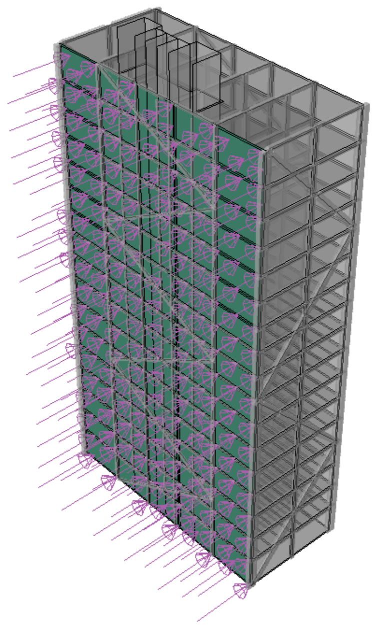


Figure 3.12: Application of pressure load

It should be noted that the method of applying wind load as a time series is only implemented in a simple form, and some tweaking of the python scripts should be expected to achieve the desired performance. For instance, the loading is currently only applied in the transverse direction and the direction can not be changed in the Excel input file, it has to be changed through editing the script. The magnitude of the load (i.e. the factor who scales the amplitude) also has to be defined directly in the code, or changed in the Abaqus GUI after the model is generated. It should also be considered to apply the load as line loads directly on the frame structure to avoid excessive local deformations of the wall panels (depending on the specified stiffness/thickness of the connector zones and the wall panels it self).

3.11 Materials

The user is able to define as many materials as desirable. The material defined can be either isotropic, transversely isotropic or orthotropic. The stiffness parameters are defined by engineering constants, as explained in subsection 2.1.2. The stiffness relations of the different material types are defined by two, five and nine parameters, respectively. The material density is also user specified.

3.12 Damping

Damping can be added to the structure in different ways: as material damping, as damping in connections, as damping in the foundations and as global damping for the entire model. The approach for adding damping to the foundations is explained in section 3.8. Damping in the connections is in reality material damping that is added to the material assigned in the connector-zones.

On the material and element level, damping can be added in three different ways: as Rayleigh damping, as composite damping or as structural damping. The option to have different damping parameters for the main members and the connection segments/zones assigned with the same material is made possible by the script creating a duplicate of the material, for use in the connection segments. The copy of the material is identical with the original material except for the damping parameters. The copy is made automatically, without user action. The different damping types are explained in short below, for further information the reader is encouraged to read the Abaqus documentation [29], especially section 2.5.4 *Damping options for modal dynamics*.

- Rayleigh damping is defined by the two factors α and β (or α_0 and α_1) who relates the element damping matrix to the mass and stiffness matrices respectively. The damping matrices for each element is then assembled to a system damping matrix. The Rayleigh damping is viscous, i.e. the damping force is proportional to the velocity. Rayleigh damping also discussed in subsection 2.2.3.

- Composite damping is defined as a fraction of the critical damping for each material. The values defined for each material is converted into mass weighted values for the modes specified to include composite modal damping on the global/modal level. The composite damping is also viscous, like the Rayleigh damping.
- Structural damping differs from composite and Rayleigh damping in the way that it is proportional to the forces in the structure instead of being velocity proportional. Due to this, structural damping is probably the most accurate way of modelling damping in timber structures, however Abaqus has some restrictions to when structural damping can be used (assumes velocity and displacements 90° out of phase) [29].

On the modal/global level direct modal, composite modal and Rayleigh damping can be specified.

- The Rayleigh damping on a global is similar to the Rayleigh damping on a local level. α and β values are defined for selected modes and applied to the entire structure.
- Direct modal damping allows for damping ratios to be applied directly to one or more modes of vibration.
- The composite damping on the global/step level is strongly related to the composite damping on the material level. The damping ratios that are defined for each material are converted into damping ratios on the global level, using the following equation from section 2.5.4 in the Abaqus documentation [29].

$$\zeta_\alpha = \frac{1}{m_\alpha}\phi_\alpha^M \zeta_m M_m^{MN} \phi_\alpha^N \tag{3.1}$$

where:
ζ_α = Damping ratio in mode α
ζ_m = Damping ratio for material m
M_m^{MN} = Mass matrix of material m
ϕ_α^M = Eigenvector corresponding to mode α
m_α = Generalized mass associated with mode α

3.13 Analysis Steps

Before running an analysis in Abaqus, one or more analysis steps must be defined. An analysis step is connected to a certain type of analysis procedure, such as a static analysis, eigenvalue/vector extraction, dynamic time-domain analysis etc. The following analysis steps are currently implemented in the scripts of the parametric model:

- The first step is a general static step where gravity is applied, more loads can be added manually in Abaqus after generating the model or by modifying the script. The step is implemented in the script as a linear step.
- The second step is a frequency step used to extract the natural frequencies and mode shapes of the structure. The frequency step is also necessary for the upcoming modal dynamics steps.
- The third step is called "Free Vibration" and is a modal dynamics step. The purpose of this step is to determine the logarithmic decrement of the building, caused by the different damping methods applied to the model (ref. section 3.12). An impulse load is applied at the top of the building in the wind direction specified in the wind-load section of the input file. The building is then allowed to freely vibrate and the logarithmic decrement is calculated later in the script based on the magnitude of the peaks. The free vibration step is also used to determine the first natural frequency (in the wind direction) for the wind calculations. This step is by default deactivated when using the *TTB_3D.py* script for running the analysis, however it is a important part of the procedure programmed in the *TTB_3D_EC_Wind.py* script.
- The fourth step is also a modal dynamics step. In this step a dynamic pressure load is applied to one of the walls, the load amplitude is defined in a .txt file that can be changed by the user (ref. subsection A.2.18). This step may be appropriate to use for analyzing the response of the structure to a specific time-history of wind loading, either measured or generated from a wind spectrum. In this thesis this step is not used for anything, however it is implemented such that it may be put to use later.
- The final step is a static step used to calculate the response (deflection) of the structure to wind load according to the method given in Eurocode 1 - part 1.4 [23]. See subsection 2.3.3 and subsection A.2.17 for more information on the wind calculations. As for the free vibration step this step is only a part of the procedure in the *TTB_3D_EC_Wind.py* script.

Additional analysis steps can be added manually in GUI of Abaqus CAE, if needed.

Chapter 4

Case Study: Mjøstårnet

This chapter explains how the different input to the parametric model were set to recreate "Mjøstårnet" [30][31]. A brief overview of the structural system can be found in section 2.5. The modelling is based on Revit- and IFC-models provided by Sweco, in addition to drawings of different components, provided from their respective suppliers. Some of the information used in the modelling process is confidential, and thus can not be presented in detail in this chapter. However, the input file created *Basemodel_input.xlsx* is available in the digital appendix. The input that is not discussed in this chapter is taken directly from one of the sources. The "base-model" established in this chapter is later used for a sensitivity study presented in chapter 5. The base-model and the results from the sensitivity study are then used to pick a few important parameters which are further improved by the use of a model updating technique (chapter 6), with the objective of making the model behavior as close to the real life behavior of the tower as possible.

4.1 Frame

The material used for the frame of Mjøstårnet, including the diagonals, is glulam with strength classes GL30c and GL30h [31]. Simplifications in the model, does not allow for detailed material specification for single members, and since GL30c is the strength class most prevalent in the building, it was decided to assign GL30c to all frame members. The influence of using GL30h in certain members was considered to have negligible effect on the dynamic behavior of the building. The material used for the model is defined as transversely isotropic, and the material parameters are in accordance to NS-EN 14080:2013 [32].

As explained in section 3.3, predefined groups of cross-sections are defined in the model. This puts some restrictions to the modelling of Mjøstårnet. In the real structure, some of the columns are tapered towards the top. The cross-sections of the internal beams also vary depending on the kind of floor they are supporting as well as their span length. Neither of these variations are included in the base model, where all members of a column group have the same cross-section along their entire lengths, and only one cross-section is assigned for all internal beams. Instead the cross-sectional groups are represented by a reference cross-section.

The frame is modelled using B32 elements, an element meant for three dimensional models, that uses quadratic polynomials for interpolation of the displacements [28]. The elements use Timoshenko theory, and therefore includes the effects of transverse shear deformation. This is especially required for connection segments, as they can have a relatively high height to length ratio.

The most uncertain part of the frame, is the connections. Preliminary calculations of connection stiffness were not conducted. For the base model a fraction of 0.2 of the original cross section area/2nd moment of area was applied to all the connections. The length of the connection zones is set to be equal to the largest dimension of the original cross-section for the respective group. The limit for the use of Timoshenko elements is when the cross-section dimension is approximately 1/8 of the element length. If the ratio is greater, the accuracy of the results are no longer guaranteed [28].

Using a segment length equal to the largest dimension of the original cross-section, allows for accurate results for a connector cross-section with a dimension that is somewhere between 10-20% of the original cross-section. This raises the question if the modelling approach for the connections is suitable for relatively stiff connections, as the segment length would have to be very large for these connections in order to satisfy the limitations of the Timoshenko theory. This possible source of error will not be studied further in the thesis, and it is assumed that the modelling approach produces result with adequate accuracy.

4.2 Floors

"Mjøstårnet" uses a combination of 300 mm concrete floors in the upper levels (levels 12-18) and prefabricated timber elements in the lower levels (levels 2-11).

4.2.1 Timber Floor Elements

The timber floor elements are fabricated by Moelven and is a part of their "Trä8" system. "Trä8" is Moelvens system of prefabricated structural elements including columns, beams, floors and bracing, designed for relatively large buildings with spans up to 8 meters. A typical floor element used in Mjøstårnet is 2.4 meters wide and spans around 7 meters. The upper flange is a Kerto-Q LVL (laminated veneer lumber) plate, and is covered by a acoustic panel and a thin layer of cast in-situ concrete after installation. The bottom flanges consist of multiple pieces of structural timber and is not continuous over the width of the elements. Between the upper and lower flanges the web is of glued laminated timber (Glulam), with some stiffening members of Kerto-S LVL placed perpendicular to the span direction. See Figure 4.1 for an illustration of a floor element.

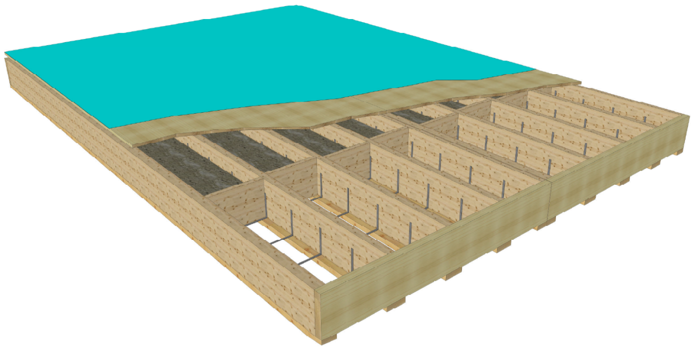


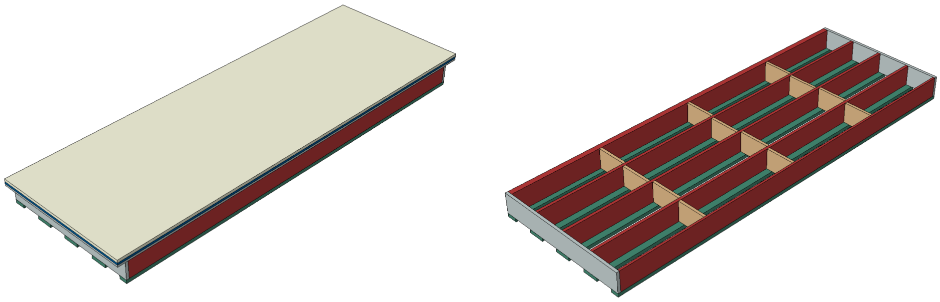
Figure 4.1: Trä8 floor element, figure from [40]

Detailed modelling of every single floor element in the global finite element model would probably be the most accurate approach, but also very complex, inefficient and computationally demanding. The intended use of model is to study the overall performance of the system, and thus a simpler approach to the modelling of floors is deemed sufficient. Instead a detailed model of a single floor element was made in Abaqus using a fine mesh of solid elements to achieve high accuracy. The model is shown in Figure 4.2, different colors indicate different materials. The material data used in the model are mean values (see Table 4.1), and are acquired from Metsä Wood [41], CEN [11] [12] [32] and Nesheims script [42].

Loads were applied separately in all three directions and the resulting deformations were measured. Then the floor element was modelled using a simple shell element, see Figure 4.3. Identical loads and boundary conditions equivalent to those of the solid model were introduced. Then an optimization routine made in Simulia Isight [37] were used to find the combination of the material parameters E_1, E_2, E_3 and the section height that gives deformations similar to the results

Table 4.1: Material data used in Abaqus model. Density and stiffness are of units kg/m^3 and MPa respectively.

Material	ρ	E_1	E_2	E_3	ν_{12}	ν_{13}	ν_{23}	G_{12}	G_{13}	G_{23}
Kerto-Q	510	10500	2200	130	0.11	0.81	0.7	820	430	22
Kerto-S	510	13800	450	130	0.61	0.74	0.6	600	600	11
C24 Timber	420	11000	370	370	0.39	0.49	0.64	690	690	30
GL32C	450	13700	460	460	0.39	0.49	0.64	850	850	30
Acustic Plate	250	162	162	162	0.3	0.3	0.3			
B30 Concrete	240	26600	26600	26600	0.2	0.2	0.2			



(a) With LVL flange, acoustic plate and concrete layer

(b) Upper flange removed

Figure 4.2: Detailed Abaqus model of Trä8 floor

from the detailed model. To limit the number of unknown variables the Poisson ratios where set to 0, and all shear moduli was given fixed values of 130 MPa. The results from the optimisation are presented in table 4.2. It can be seen that the error is relatively low, hence the shell element can be used as a relatively good approximation to the floor. Using the simplified shell has numerous benefits, with the most important being a significant reduction in computational time due to the reduction in dofs. In addition to much simpler modelling (or faster model generation when using parametric modelling) and less sources of error when it comes to e.g. boundary conditions.

Table 4.2: Isight results

						Relative Error		
Run	t	E_1	E_2	E_3	Objective func.	Lengthwise	Transverse	Out of plane
Initial	0.4	2600	1300	1300	7.5239	−5.2273	−0.0232	−2.5785
Best	0.238	59866	100	10352	0.0095	0.0304	−0.0444	−0.0304

The timber floors are modelled with connection-zones in order to simulate connections between the element. In reality, the width of the elements varies, but the script only allows for a single element width. An element width of 2.4 m was deemed to be representative. The width of the connector zones were set to 250 mm, equal to the thickness of the original cross-section. The material used in the zone is also similar to the material used for the timber floor elements. A thickness

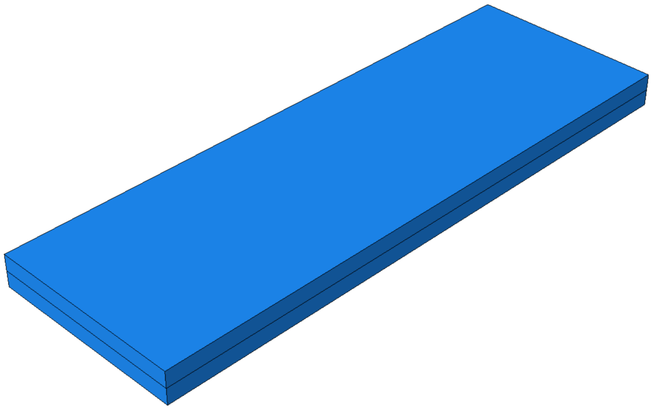


Figure 4.3: Trä8 floor modelled using shell elements

fraction of 0.1 was chosen as a starting point. This value is not based on any calculations, and is purely an initial guess. The connection-zones for the floor-to-shaft connections are assigned with the same properties.

4.2.2 Concrete Floors

The concrete floors are assumed uncracked, and can therefore be modelled by an isotropic material with the properties as shown in Table 4.3. The properties are taken from Nesheim [42]. The thickness of the floors is 300 mm. The concrete floors are modelled as continuous shells, as they are partly cast in place.

Table 4.3: Material properties used for the concrete floors

Parameter	Density	Young's Modulus	Poisson's ratio
Value	2400 kg/m^3	26 600 MPa	0.2

For the floor-to-shaft connections of the concrete floors, the thickness ratio is set to 0.1. The width of the connection zone is set to 300 mm, the same as the thickness of the original floor. The material used in the connection zone is the same as for the concrete floors.

4.3 Walls

4.3.1 Shaft Walls

Since the shaft walls are loaded almost exclusively in-plane, lower accuracy of the out of plane bending stiffness is accepted. This makes it possible to make the following simplification (Equation 4.1), if we assume that only the layers oriented in the direction of the load contributes to the stiffness:

$$E_{eq,i} = E_{CLT} \cdot \frac{A_i}{A_{Tot}} \tag{4.1}$$

where:

E_{CLT} = The Young's modulus of a CLT lamella/layer in its span direction
$E_{eq,i}$ = The equivalent Young's modulus in direction i
A_i = The total cross sectional area of layers with span direction i
A_{Tot} = The total cross sectional area of all layers

As a as a result of the aforementioned the CLT panels can be modeled using a homogeneous shell section with thickness t, $E_{eq,1}$ and $E_{eq,2}$ as the Young's moduli in-plane and E_3 out of plane. The material parameters are from Unterwieser and Schickhofer [43], based on CLT with bending strength of 24 MPa [31]. Detailed data about the thickness and number of layers of the CLT walls were not available, but pictures from Abrahamsen [31], showed that cross-sections with both three and five lamellae were used. As explained in section 3.6, it is only possible to assign one section to all shafts of the model. It was decided to use a cross-section of five lamellas in the model of Mjøstårnet. The parameters used are given in Table 4.4.

Table 4.4: CLT - Modeling Parameters

Parameter	N_{Layers}	Thickness	Density	E_{CLT}	$E_{eq,1}$	$E_{eq,2}$	E_3
Value	5	150 mm	420 kg/m^3	11 600 MPa	6960 MPa	4650 MPa	300 MPa

Note that it was decided to attach the shaft walls to the building, contrary to how the building was designed. The reason for doing this is that it is believed that even though it was not designed for it, some stresses will be transferred from the floors to the shaft. The connection-zones around the shafts was intended to simulate the real behaviour of the structure.

4.3.2 Exterior Walls

The exterior wall panels is a more complex structure than the CLT to represent using shell elements. Ideally a similar approach as for the prefabricated floor elements (subsection 4.2.1) could be taken, with detailed modeling of a single module followed by tweaking the thickness and material parameters of a shell to recreate the results of the detailed model. However, lack of detailed drawings, as well as limited time, lead to the wall stiffness being determined by engineering judgement combined with trial and error.

The stiffness-contribution from the wall panels was assumed to be very low due to the way it is connected to the building (and the fact that they were left out of the original FEA model by Sweco), but still high enough to avoid local modes with low frequencies. With that in mind a thickness of 0.450 m and a isotropic material with a Young's modulus of $2 \times 10^7\,\mathrm{N/m^2}$ were chosen for the exterior walls. The density was set to $250\,\mathrm{kg/m^3}$. This is a rough estimate based on the density for a timber framing exterior wall from Byggforskserien [44].

4.4 Live Loads and Additional Mass

The pergola placed on the roof of Mjøstårnet, is included only as a non-structural mass in the model. This is done for sake of simplicity, as including the option for adding architectural elements in the script would be difficult to generalize. However, it would be possible to include the pergola as part of the frame by adding single members to the frame. Even so, the pergola was considered to have little influence on the structure other than its mass contribution, and is therefore represented as a uniformly distributed load acting on the roof in the model. The weight of the pergola was converted to an equivalent distributed load of $101.3\,\mathrm{kg/m^2}$.

The balconies of Mjøstårnet are also assumed to have no impact on the structural performance, and are thus represented by non-structural point masses. The weight of the balconies were roughly estimated to be 2500 kg, including live load.

Eurocode 1 part 1-1 [45] states values for the imposed loads on the structure. The imposed loads are depending on the intended usage of the the respective area, and includes things like people, furniture etc. For Mjøstårnet the usage categories and characteristic loads listed in Table 4.5 were identified.

To account for the fact that the the areas are not loaded with the full magnitude of the imposed loads at all times, the quasi-permanent load combination for Euro-

Table 4.5: Imposed loads

Usage	Category [(1)]	Levels	q_k [(2)]
Offices	B	$0-6$	$3.0\,\mathrm{kN/m^2}$
Hotel	A	$7-10$	$2.0\,\mathrm{kN/m^2}$
Apartments	A	$11-16$	$2.0\,\mathrm{kN/m^2}$
Rooftop Terrace	A	17	$4.0\,\mathrm{kN/m^2}$

(1): Table NA 6.1 in [45] (Norwegian annex)

(2): Table NA 6.2 in [45] (Norwegian annex)

code 0 [46] were used and the resulting loads were converted into a equivalent distributed mass by dividing the distributed load by the gravitational acceleration, $g = 9.81\,\mathrm{m/s^2}$. The resulting masses are listed in Table 4.6:

Table 4.6: Distributed mass

Usage	Ψ_2 [(1)]	Dist. Mass
Offices	0.3	$91.8\,\mathrm{kg/m^2}$
Hotel	0.3	$61.2\,\mathrm{kg/m^2}$
Apartments	0.3	$61.2\,\mathrm{kg/m^2}$
Rooftop Terrace	0.3	$122.4\,\mathrm{kg/m^2}$

(1): Reduction factor - Table NA.A1.1 in [46]

4.5 Finite Element Types

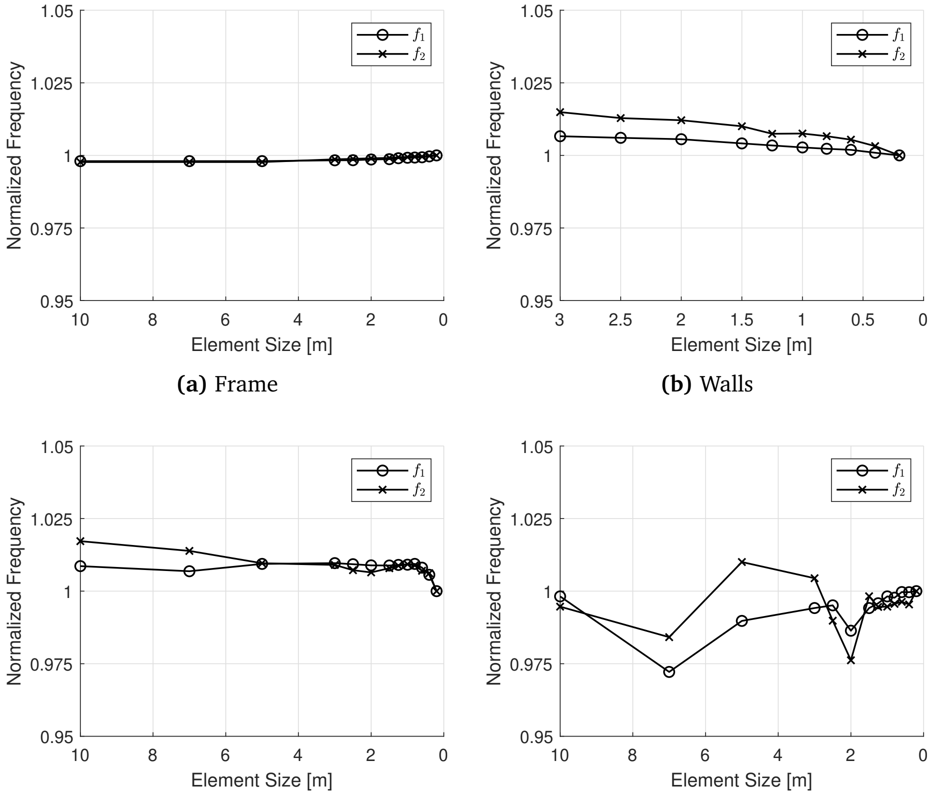
The element types used in the different parts of the model are presented in Table 4.7. The element types are chosen based on efficiency, while still retaining good accuracy.

Table 4.7: Elements used in the finite element analysis of Mjøstårnet

Part	Element Type	Description
Frame	$B32$	3-noded quadratic "Timoshenko" beam element
Floors	$S4R$	Quadrilateral 4-noded element with reduced integration
Exterior Walls	$S4R$	Quadrilateral 4-noded element with reduced integration
Shaft Walls	$S4R$	Quadrilateral 4-noded element with reduced integration

4.6 Convergence Study

Prior to the main part of the parameter study, the convergence of the FEA-model (see section 2.4) was checked to ensure that the output parameters (eigenfrequencies) is of sufficient accuracy. The element size were changed in steps ranging from ≈ 10 m to ≈ 0.1 m. The smallest mesh size is assumed to be the most accurate but the number of elements and nodes becomes large and the calculation extremely inefficient.



(a) Frame **(b)** Walls

(c) Floors **(d)** Shafts

Figure 4.4: Convergence of different parts. Frequencies are normalized w.r.t the most accurate mesh.

The study (Figure 4.4) found that an element size of $1m$ ensures high accuracy (less than 1% deviation from the most accurate mesh) while being significantly faster than the finer mesh. Note that only the convergence of the two first eigenfrequencies are studied, if e.g. higher vibration modes or stresses/strains were to be studied the convergence is typically much slower. Note that the analysis failed when the element size was set to >3 m in the walls Figure 4.4b. Also the convergence of the shafts are somewhat doubtful in the way that it is clearly not monotonic, however since the deviation in the results are relatively small it is still deemed acceptable for the purpose of this study.

4.7 Simulation Results

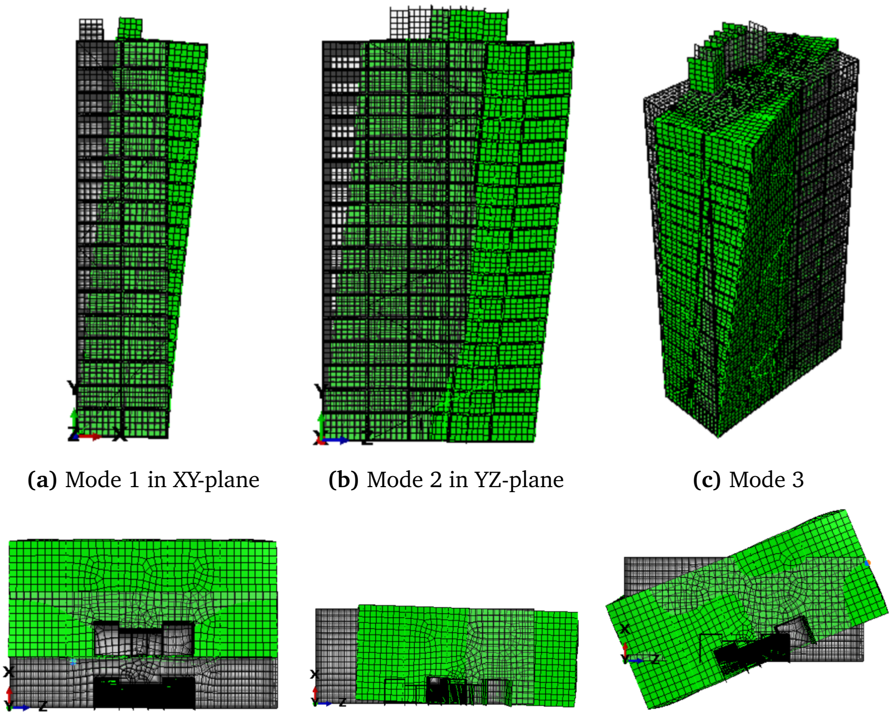
The results from running a simulation using the input described in this chapter is presented in Table 4.8 and Figure 4.5. The results only show the first three fundamental modes, as the higher-order modes are dominated by local modes. These local modes may occur due to weaknesses in the model, and will therefore not be found in the real structure. Mode 1 is bending in the transversal direction. Mode 2 is mainly bending in longitudinal direction. However, as seen in Figure 4.5e, the mode also includes some torsional movement. Finally, mode 3 is purely a torsional mode.

In Table 4.8 the results from the base model are compared to the measurements based on ambient vibrations (subsection 2.5.3) and the frequencies produced by the numerical model by Sweco (subsection 2.5.2). The base model produce frequencies that are considerably lower than what is measured. A deviation from the measurements was expected due to many of the input parameters in the base model being highly uncertain. However, the mode shapes are similar in terms of direction. It is at the time of writing not possible to study the exact mode shapes of finished building, due to limitations in the monitoring equipment.

The numerical model developed by Sweco produces frequencies that are lower compared to the base (parametric) model. The higher frequencies in the parametric model, is likely to be due to exterior walls being included and the shaft being connected to the rest of the building. Neither of which are included in the other numerical model. More importantly, the mode shapes produced by the two models differ (see Figure 2.13). By visual verification, it can be seen that mode 1 and 2 are opposite in the two cases, while mode 3 is similar for both models. Since the main difference between the two models is the inclusion of shafts and exterior walls in the base model, it is likely that the change of modes is linked to this.

Table 4.8: Fundamental frequencies of base model compared to measured frequencies and results from numerical model used for design.

Frequency Nr.	Base Model	Measured	Numerical Model by Sweco
f_1	0.39 Hz	0.50 Hz	0.33 Hz
f_2	0.41 Hz	0.54 Hz	0.37 Hz
f_3	0.63 Hz	0.82 Hz	0.59 Hz



(a) Mode 1 in XY-plane **(b)** Mode 2 in YZ-plane **(c)** Mode 3

(d) Mode 1 in XZ-plane **(e)** Mode 2 in XZ-plane **(f)** Mode 3 in XZ-plane

Figure 4.5: The first three fundamental modes of the base model. The grey parts show the undeformed geometry.

Chapter 5

Sensitivity Study

A sensitivity study is performed to determine the influence of a selection of the input parameters on the model output. As a starting point the input of the parametric model described in chapter 3 was set to imitate Mjøstårnet (see chapter 4 for details on the base setup), the excel workbook containing the input used is also provided in the digital appendix. Then the variables were changed (one by one) in relatively small steps in intervals chosen independently for each variable. Simiulia's software Isigth including the DOE (Design of Experiment), Excel and Simcode components was used to update the parameters and run the analyses (see Figure 5.1), post processing was done in Matlab.

The frequencies of the three first vibration modes were chosen as the output variables in this study. The two first modes are bending modes in the x- and z-direction respectively, and the third is a torsional mode rotating about the height (y-) axis of the tower. The reason for limiting the sensitivity study to only three modes, is that preliminary tests showed that only these three are consistent for a variety of different parameters. Higher global modes will be swapped with local modes with low frequency for certain parameter inputs, and can therefore not be studied. Input parameters studied includes variables such as axial and rotational stiffness of connections, foundation stiffness, material parameters etc.

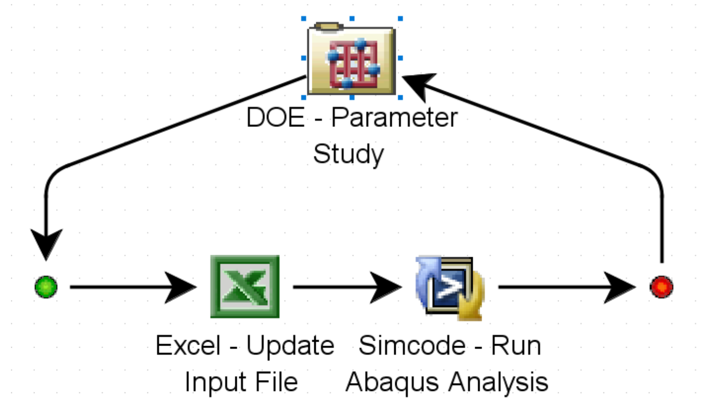


Figure 5.1: Isight setup for a parameter study

5.1 Vertical Stiffness of Foundation

The model is equipped with springs at the end of each column to simulate the stiffness of the foundation. The ground conditions at the site of Mjøstårnet were challenging and there is a huge amount of uncertainty related to the stiffness. A large span of spring stiffness values, ranging from 1×10^8 N/m to 2×10^9 N/m per spring, were analysed in the sensitivity analysis due to the high level of uncertainty.

The analysis showed that the vertical stiffness has great impact on all three frequencies. The tower is basically a cantilever beam clamped to the ground. When the stiffness is low, the tower will rotate at it's base while the tower will act as a rigid body. As the stiffness increases the base rotation will be reduced and the mode will be gradually more depending on the tower bending. The first mode of the tower are showed in Figure 5.3, with low and high foundation stiffness respectively.

Another interesting observation that can be seen in Figure 5.2 is that the gap between the first two frequencies, i.e. the bending modes, are decreasing as the foundation stiffness is increasing. In fact, another analysis with even higher stiffness confirmed that if the stiffness gets high enough the direction of the two first modes will switch, i.e. what has previously been mode 1 will become mode 2 and vice versa. The cause of this is most likely that the length of building in the direction of first mode the are less than half the length in the direction of the second.

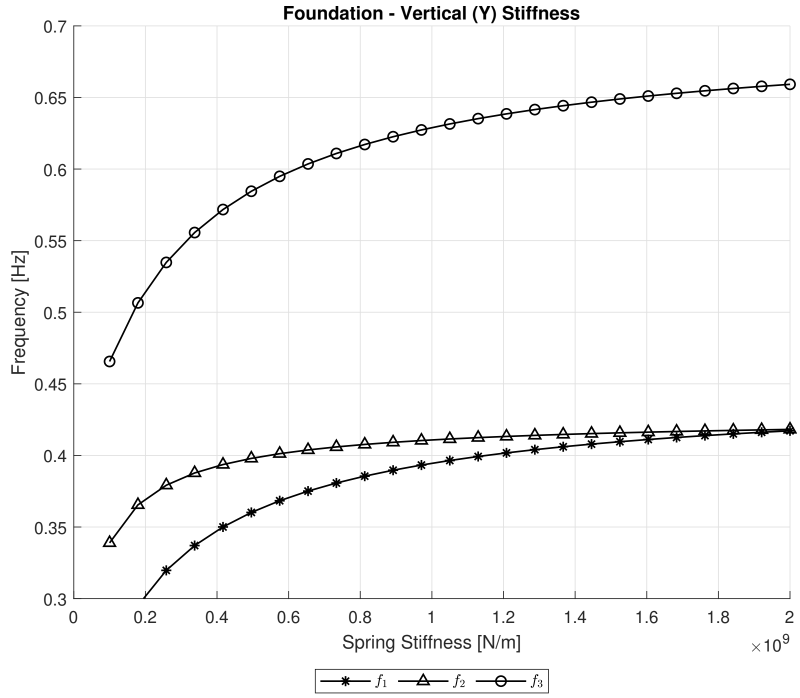
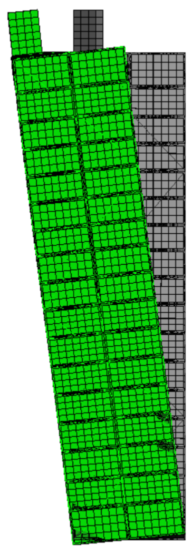
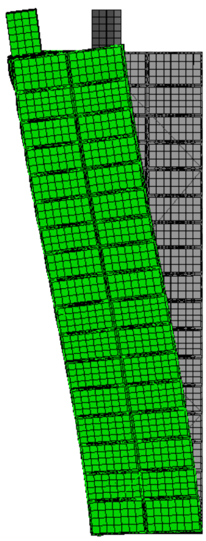


Figure 5.2: The three first eigenfrequencies as a function of vertical foundation stiffness



(a) Soft foundation

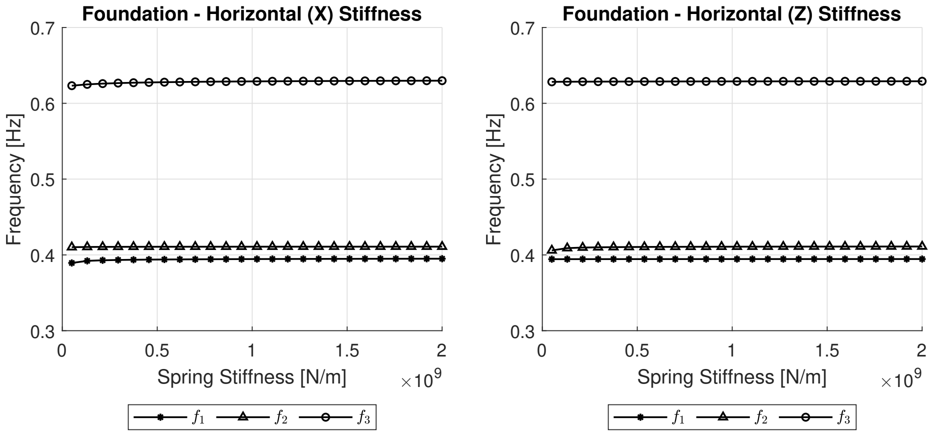


(b) Stiff foundation

Figure 5.3: The first mode shape of the tower with high and low foundation stiffness. Notice the difference in base rotation and bending.

5.2 Horizontal Stiffness of Foundation

Similarly to the vertical direction (section 5.1) the model also has the possibility to independently change the spring stiffness in the two orthogonal horizontal directions. Due to the mainly vertical orientation of the piles used as foundation, it is assumed that the horizontal stiffness is considerably lower than the vertical stiffness. The analysis is therefore ran in the interval from 5×10^7 N/m to 2×10^9 N/m, results in Figure 5.4).



(a) X-Direction **(b)** Z-Direction

Figure 5.4: The three first eigenfrequencies as a function of horizontal foundation stiffness

The horizontal foundation stiffness has little influence on the frequencies, except for when the stiffness is very low ($\lessapprox 10^8$ N/m). As for the vertical springs (section 5.1), low stiffness causes the modes to be dominated by rigid body motion. Hence for the first and second mode who are translational modes, the building will "slide" sideways at the base. While for the torsional mode, the building will rotate at its base.

5.3 Rotational Stiffness of Foundation

The final test performed on the foundations is an analysis of the effects of the rotational stiffness of the support of each individual column. The rotational stiffness tested ranged from 0 N/rad to 10^{15} N/rad, meaning that its tested from completely free to rotate, up to a such a high stiffness that it is effectively rigid (magnitudes stiffer than the column it self). To limit the amount of test, all three rotational degrees of freedom were changed at the same time.

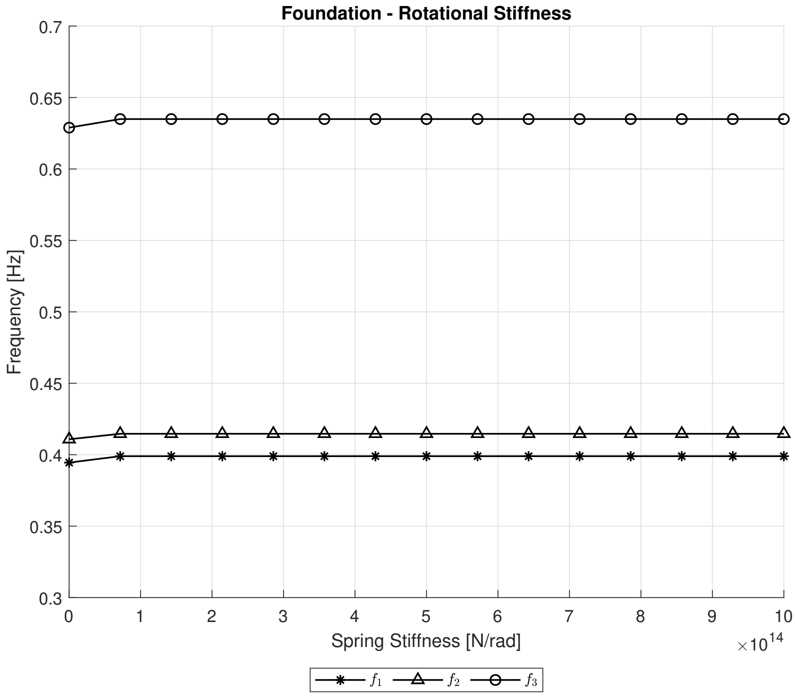


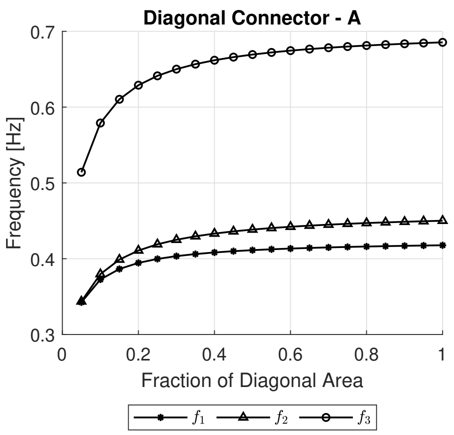
Figure 5.5: The three first eigenfrequencies as a function of the rotational stiffness

Figure 5.5 shows that changing the rotational stiffness is of little significance for the three lowest frequencies. A small change can be spotted at the lower end of the interval, but it is tiny compared to the changes caused by e.g. changing the vertical stiffness (section 5.1).

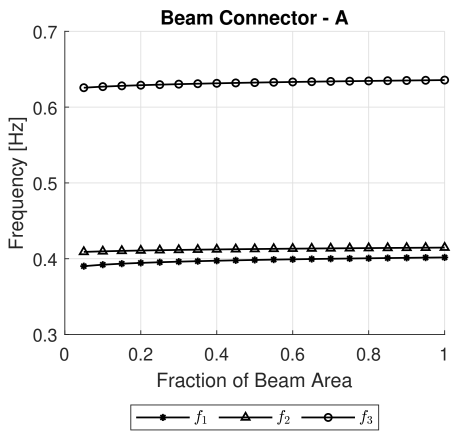
5.4 Axial Stiffness of Connections - Frame

To study the influence of axial stiffness in the connections on the eigenfrequencies, the area of the beam/diagonals are reduced in a segment near each connection. The modelling choices are discussed further in section 3.7. The area in the connector zone is adjusted in a interval from $\frac{A_{Connector}}{A_{Original}} = 0.05$ to $\frac{A_{Connector}}{A_{Original}} = 1.0$, where $A_{Original}$ and $A_{Connector}$ is the area of the main part of the beam/diagonal and the area of the connector element respectively. The effect of reducing and increasing the area of the connector were done separately for diagonals (Figure 5.6a) and beams (Figure 5.6b)

Figure 5.6a shows that the eigenfrequencies are highly dependant on the axial stiffness of the connections connecting the diagonals to the rest of the structure.



(a) Diagonal



(b) Beams

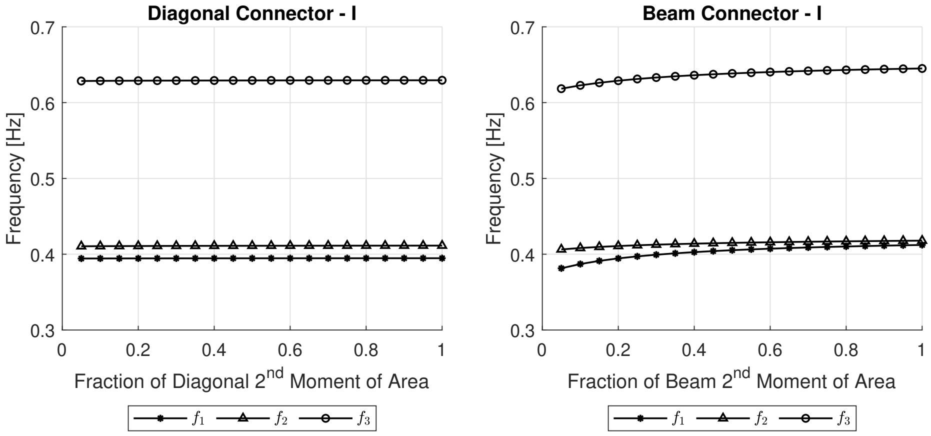
Figure 5.6: The three first eigenfrequencies as a function of connector cross section area

This is due to the truss like structural system of Mjøstårnet, which relies heavily on axially loaded diagonals for providing the horizontal stiffness of the building. As a result of the large diagonals making up most of the horizontal stiffness of the building, changing the stiffness of the beam connections affects the lowest frequencies significantly less, as shown in Figure 5.6b.

5.5 Rotational Stiffness of Connections - Frame

The effects of altering the rotational stiffness of the connections are studied in a similar way as the axial stiffness. In this case the second moment of area, I, are changed instead of the area, while the Young's moduli and segment lengths are kept constant. The rotational stiffness about both the weak and the strong axis are modified simultaneously, while the torsional stiffness are assigned a fixed value and not considered any further in this thesis. The results of analyses with I_{11} and I_{22} of the connector segment in the interval between 5% and 100% of the original beam I_{11} and I_{22} are presented in Figure 5.7

From Figure 5.7a it is clear that the rotational stiffness of the diagonal connections has hardly any impact on the lower modes of the building. Again this is because the diagonals are almost exclusively subjected to pure axial loading. However, changing the rotational stiffness of the beam connections has a significant effect, especially on the first and third eigenfrequency, likely because there are more beams spanning in the direction of the first mode.



(a) Diagonal **(b)** Beams

Figure 5.7: The three first eigenfrequencies as a function of connector second moment of area

The results of this analysis combined with the results from section 5.4 clearly shows that the building relies both on axially loaded diagonals and some moment resistance in the corners of the frames to make up the horizontal stiffness, with the contribution from the diagonals being the most significant.

5.6 Stiffness of Floor to Shaft Connections

The stiffness reduction in connections between the floors and shafts are simulated by a "connection-zone" in the floors, located at the boundary of the shafts (see subsection 3.7.2). The shell thickness inside the zone is adjusted to reduce or increase the stiffness. If the thickness, and as a consequence the stiffness, of the zone become to low, local modes with low frequencies will arise and make the results of the analysis invalid (see figure Figure 5.9). An example of a false result caused by local/spurious modes can be seen in Figure 5.8, where one of the measurements of the third frequency is clearly wrong.

The sensitivity study are run with connector thicknesses in the interval from 0.4% to 100% of the original floor. Thinner than 0.4% all the frequencies would be local modes, and any thicker than 100% would not represent a realistic connection.

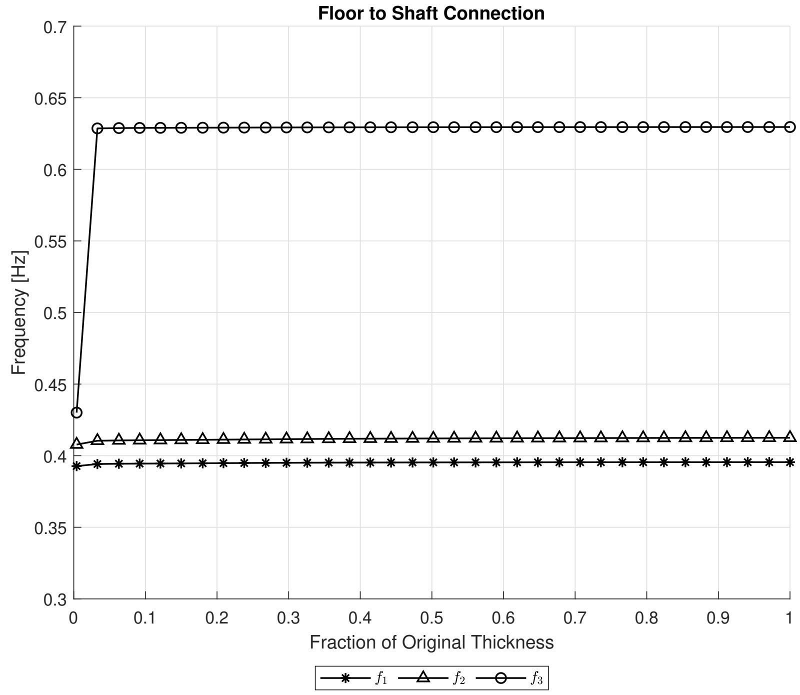


Figure 5.8: The three first eigenfrequencies as a function of shaft to floor connector thickness

Table 5.1: Comparison of analysis without ties and with low connector stiffness

Frequency Nr.	Without Ties	With Connector Zone$^{(1)}$
f_1	0.373 Hz	0.393 Hz
f_2	0.381 Hz	0.408 Hz
f_3	0.622 Hz	–$^{(2)}$

$^{(1)}$: Connector zone thickness 0.4% of original floor thickness.

$^{(2)}$: Third frequency invalid due to local modes.

In addition to the results plotted in Figure 5.8, an analysis with the ties between the floors and the shaft entirely removed were performed. The results of this analysis compared to the results with low connector stiffness are presented in Table 5.1.

Ideally there should be very little gap in the frequencies between the analysis without ties and the analysis with very low connector zone stiffness. However, as shown in Table 5.1 there is a larger difference in the frequencies than expected. A possible explanation for this is that the frequencies are highly sensitive to changes in stiffness when the stiffness of the connector are very low (i.e. lower than what is possible to model using the approach with connector zones with reduced thickness), but converges quickly as the stiffness increases.

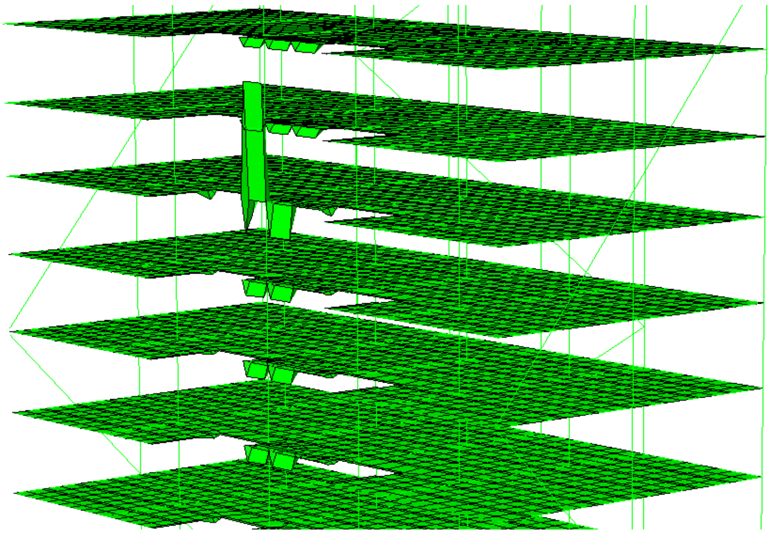


Figure 5.9: Illustration of the local modes that arise in the floor-to-shaft connection-zones when the connector thickness is set lower than 0.4%. Walls and shafts are hidden in the figure.

5.7 Stiffness of Connections Between Floor Modules

The floors who are made with prefabricated Trä8 modules (see subsection 4.2.1) are modelled with longitudinal connection zones every $\approx 2.4\,\mathrm{m}$ to represent the interface between the elements. The shell thickness of all the connection zones was adjusted in the interval from 0.1% to 100% of the original floor thickness for the sensitivity study. No local modes interfered with the frequencies of interest in the interval chosen.

All the three lines in Figure 5.10 are flat, hence the connector stiffness between the modules has no visible influence on neither of the first three modes of the structure.

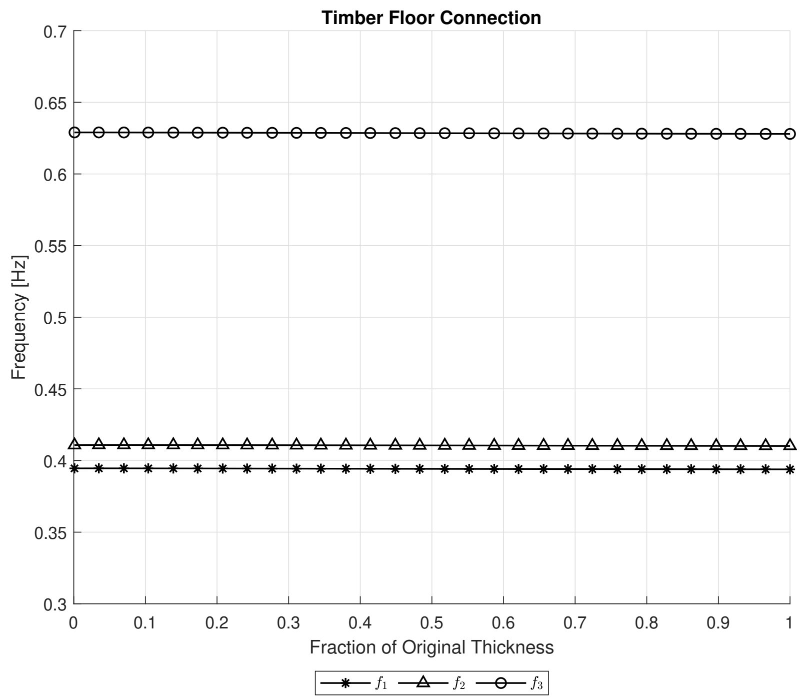


Figure 5.10: The three first eigenfrequencies as a function of floor module connector thickness

5.8 Stiffness of Wall to Frame/Floors Connection

At the outline of each exterior wall panel there are connection zones. The connection zones makes it possible to regulate the stiffness of the connection between the prefabricated wall modules and the frame structure and/or the floors. Due to the issue with local/spurious modes the lowest shell thickness possible for sensitivity study was found to be approximately 4% of the original wall thickness, while the upper limit is set to 100% as for the other analyses.

The results presented in Figure 5.11 shows that the frequencies of the second and third modes are influenced by the stiffness of the wall connections quite heavily. The first mode on the other hand remains more or less unchanged over the thickness interval tested. The same trend can be seen in the study of the material stiffness in the exterior walls section 5.11, where the second and third frequency increases with increasing material stiffness, while the first frequency seems relatively unaffected.

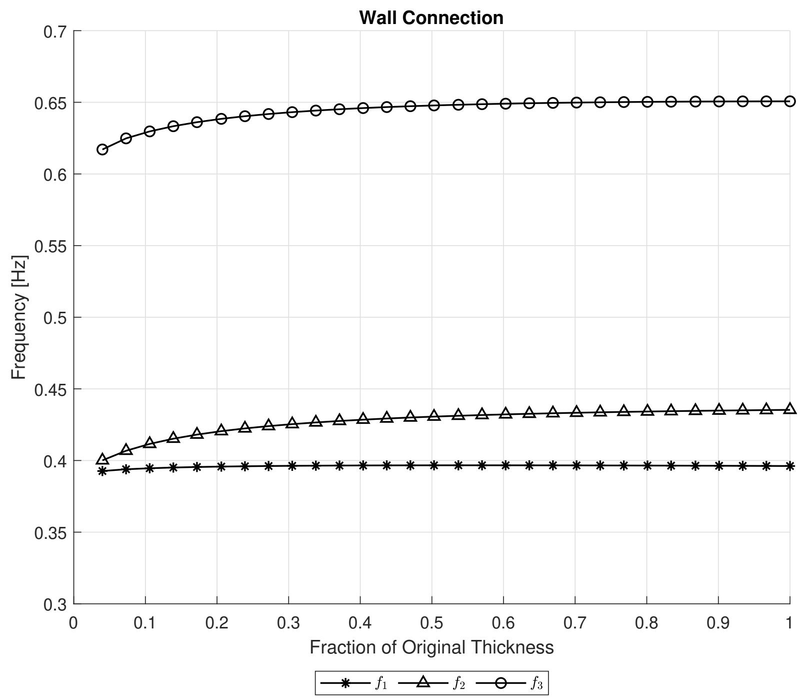


Figure 5.11: The three first eigenfrequencies as a function of wall connector thickness

5.9 Material Stiffness - Frame

The sensitivity of the fundamental eigenfrequencies to changes in the material stiffness of the glulam frame structure is tested. Timber is a natural material where the mechanical properties will vary. The Norwegian standard NS-EN 14080 [32] gives values for both mean (i.e. 50%-fractile) and 5%-fractile stiffness. The mean values are typically used for serviceability calculations, while the 5%-fractile is more conservative and used for ultimate limit state calculations. In the sensitivity analysis the mean stiffness of GL30c glulam is used as a starting point, and a total of 15 values in the interval from 70% to 130% of the mean stiffness are analysed. The interval chosen should cover all realistic values of the stiffness for the given strength class. The parallel (E_0) and the perpendicular to grain (E_{90}) Young's moduli are assumed to be dependant on the same factors (e.g. growth rate, moisture etc.), hence they are multiplied with the same coefficient and changed simultaneously for the purpose of this analysis.

In Figure 5.12 the resulting fundamental frequencies are plotted against the multiplication factor used to modify the parallel (E_0) and the perpendicular to grain (E_{90}) Young's moduli of the frame material. The graphs shows a clear, almost linear relationship between the stiffness and the frequencies. The material stiffness

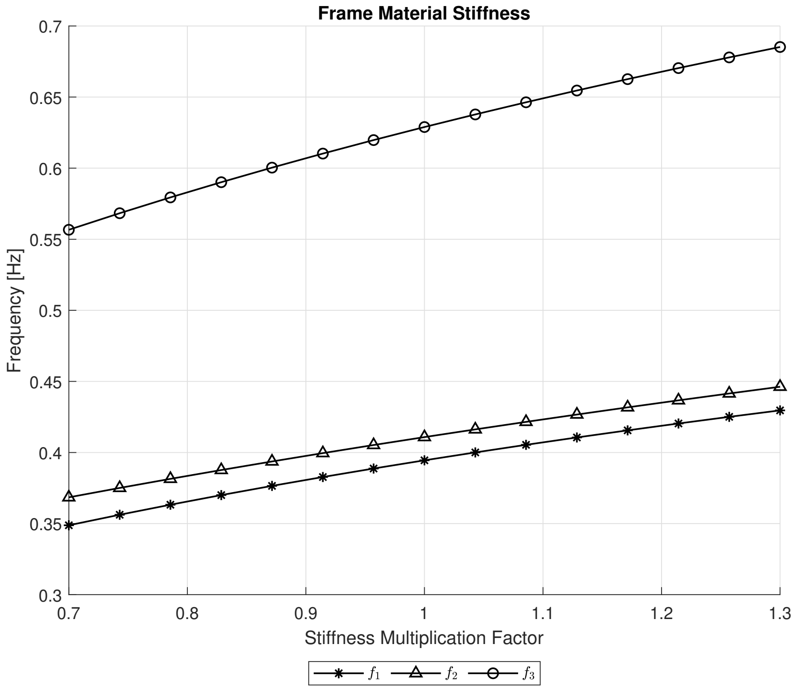


Figure 5.12: The three first eigenfrequencies as a function of frame material stiffness

of the frame is one of the most influential parameters for all of the frequencies checked.

5.10 Material Stiffness - Timber Floors

Since the main focus of this thesis is timber structures, the sensitivity analysis of the floors are limited to focus on the prefabricated timber floors described in subsection 4.2.1, while the concrete floors are left unchanged. As described in subsection 4.2.1 the composite floor elements are simplified by using shell elements with a fictitious orthotropic material with properties chosen by the use of a optimization routine. As a consequence of using a single fictitious material to represent the overall stiffness of the composite floor elements, variations in the material stiffness not only represents natural variations in the timber stiffness due to factors such as moisture content etc., but also variations in the stiffness of the interfaces (glue, nails etc.) between the different parts.

The test procedure for the floor elements are similar to the procedure described in section 5.9. However, since the material is orthotropic there are three Young's

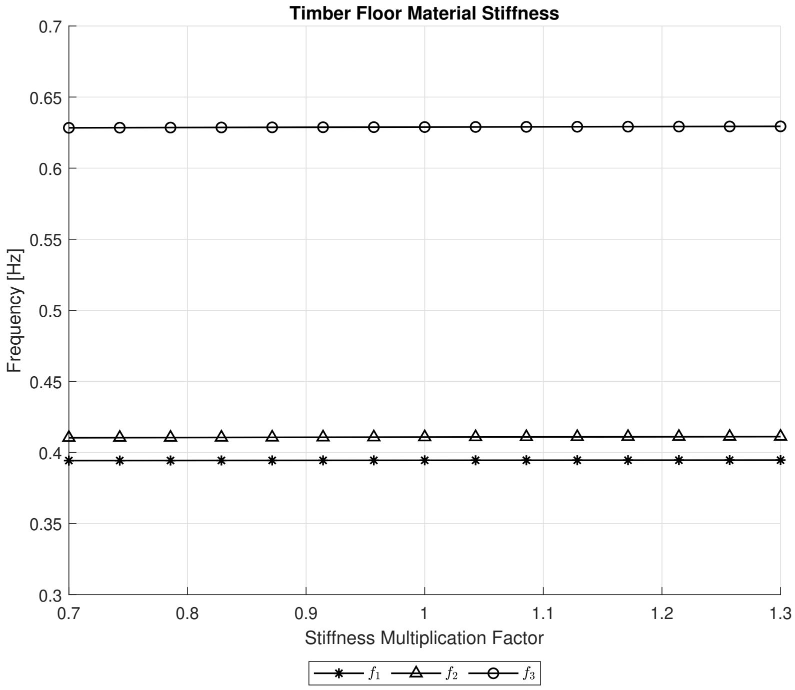


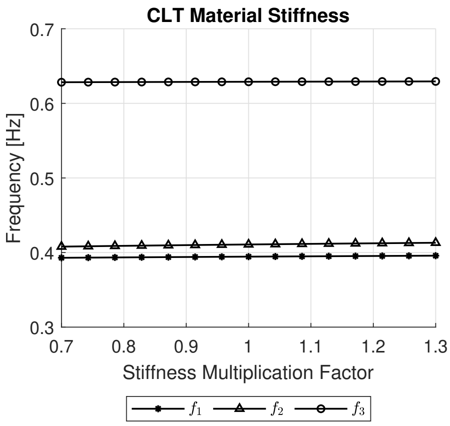
Figure 5.13: The three first eigenfrequencies as a function of timber floor material stiffness

moduli (E_1, E_2 and E_3) instead of two (E_0 and E_{90}) that are changed throughout the different steps. The results presented in Figure 5.13 shows that the stiffness of the floors are of little significance to the fundamental frequencies of Mjøstårnet.

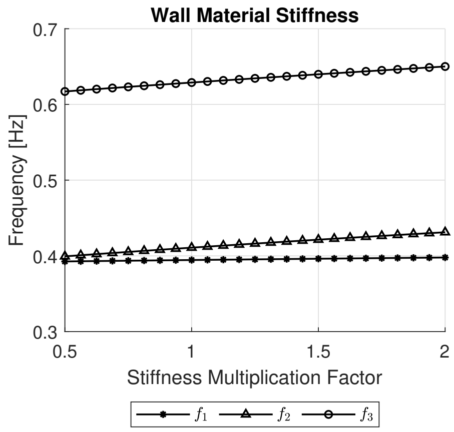
5.11 Material Stiffness - Walls

The study of material parameters are concluded with two tests performed on the walls of the tower, one on the exterior walls and one on the shaft walls. The shafts are made of cross laminated timber (CLT), while the wall panels used as exterior walls are prefabricated light frame modules made of timber.

The intervals for the sensitivity study of the walls are chosen based on the uncertainties related to the different types of walls. The variations in the material stiffness of the CLT are mainly associated with the natural variations of timber as a material. For the exterior wall however, the material used for the shell element is only a fictitious material with parameters chosen to represent the entire structure of a wall panel, including natural variation in the material, interaction between the components etc. As a consequence a larger interval is chosen for sensitivity study of the material in the exterior walls than for the other materials.



(a) CLT Shafts



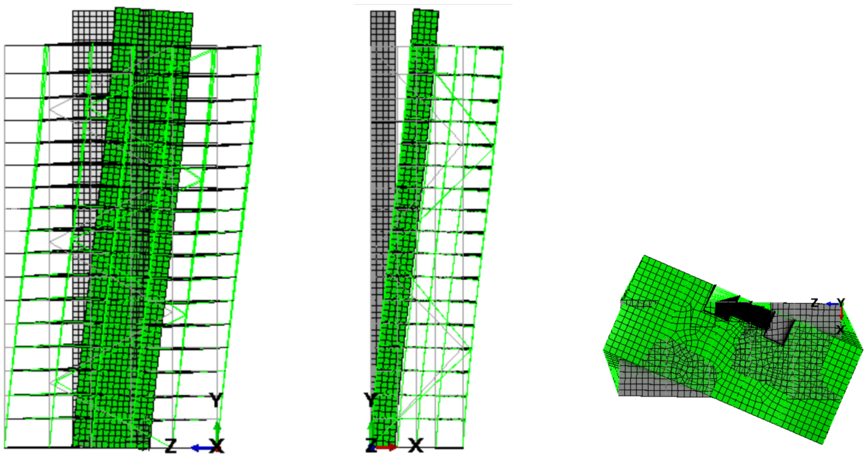
(b) Exterior Walls

Figure 5.14: The three first eigenfrequencies as a function of material stiffness

Figure 5.14a shows low correlation between the stiffness of the shafts and the eigenfrequencies of the system. This confirms the hypothesis that the shafts don't really contribute to the horizontal stiffness of the tower. The material in the exterior walls have an interesting effect on especially the second and third mode, while the frequency of the first mode remains more or less unchanged, similar to the results seen in the study of wall connections (section 5.8). If the walls are either given an even lower stiffness or left out of the model, the directions of the first and second mode will change, the same effect that can be seen with high vertical foundation stiffness (see section 5.1). Results from simulation without exterior walls are presented in Table 5.2 and Figure 5.15.

Table 5.2: Fundamental frequencies for model without exterior walls

Frequency Nr.	Frequency
f_1	0.397 Hz
f_2	0.403 Hz
f_3	0.642 Hz



(a) Mode 1 in YZ-plane **(b)** Mode 2 in XY-plane **(c)** Mode 3 in XZ-plane

Figure 5.15: Mode shapes of model without exterior walls

5.12 Summary of the Sensitivity Study

The parameter study option of Figure 5.16, 5.17 and 5.18 list all the parameters studied in the sensitivity study sorted from most to least influence on the first, second and third frequency respectively.

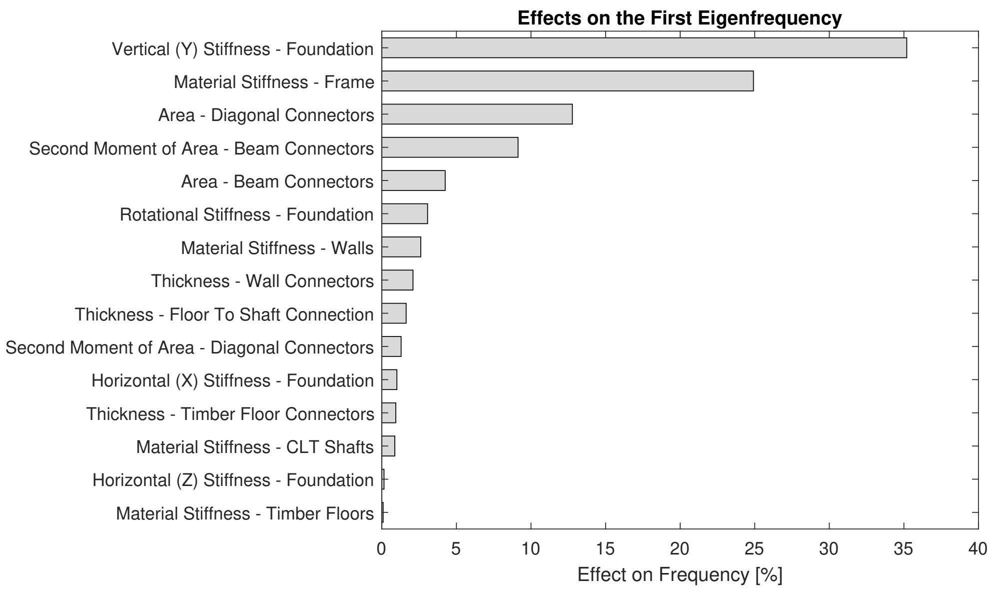


Figure 5.16: Most important parameters for the first mode

For the first frequency (Figure 5.16) the vertical stiffness of the foundation is the most important input parameter, followed by the material stiffness in the frame and the area (i.e. the axial stiffness) of the connectors in the diagonals. The least important parameters are the stiffness of the prefabricated Trä8 floor modules and the horizontal stiffness of the foundations.

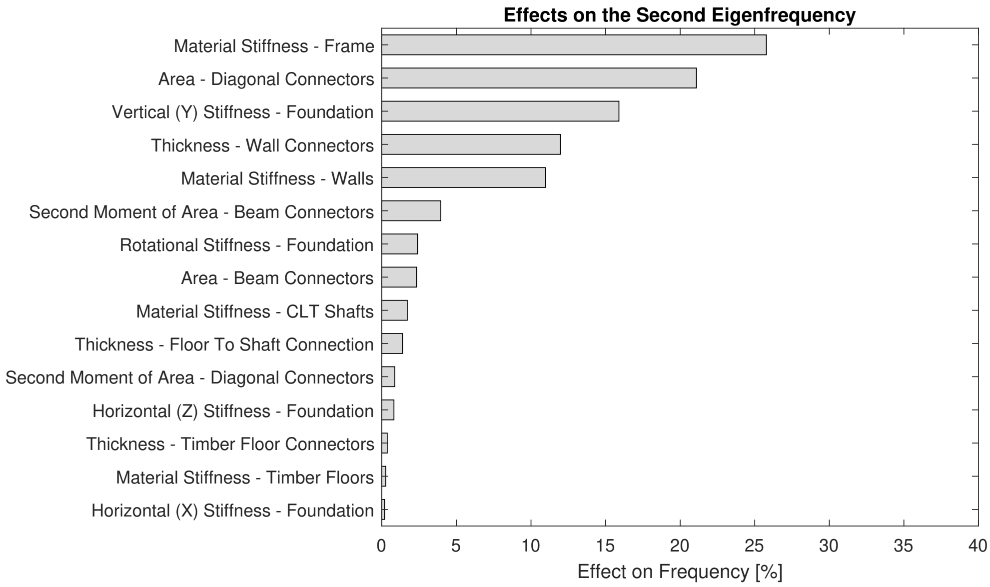


Figure 5.17: Most important parameters for the second mode

The second frequency (Figure 5.17) is highly dependant on many of the important parameters for the first frequency, albeit in a different order. Here the material stiffness in the frame are the most important, followed by the cross section area of the connector segments in the diagonals. An interesting difference is that the second frequency are more sensitive than the first to changes in the parameters (material stiffness and connector stiffness) concerning the exterior walls. As for the first frequency the parameters related to the timber floors and the horizontal stiffness of the foundation seems almost irrelevant, at least within the intervals studied.

For the third frequency (Figure 5.18), the three most influential parameters is in fact exactly the same as for the first frequency: vertical foundation stiffness, the stiffness of the frame material and the area of the connection segments of the diagonals. The least important parameters are again the stiffness of the floor modules, horizontal stiffness of the foundation, in addition to the CLT (shaft) stiffness. Similarly as for the second frequency, the exterior walls seems to be more important for the third than the first mode.

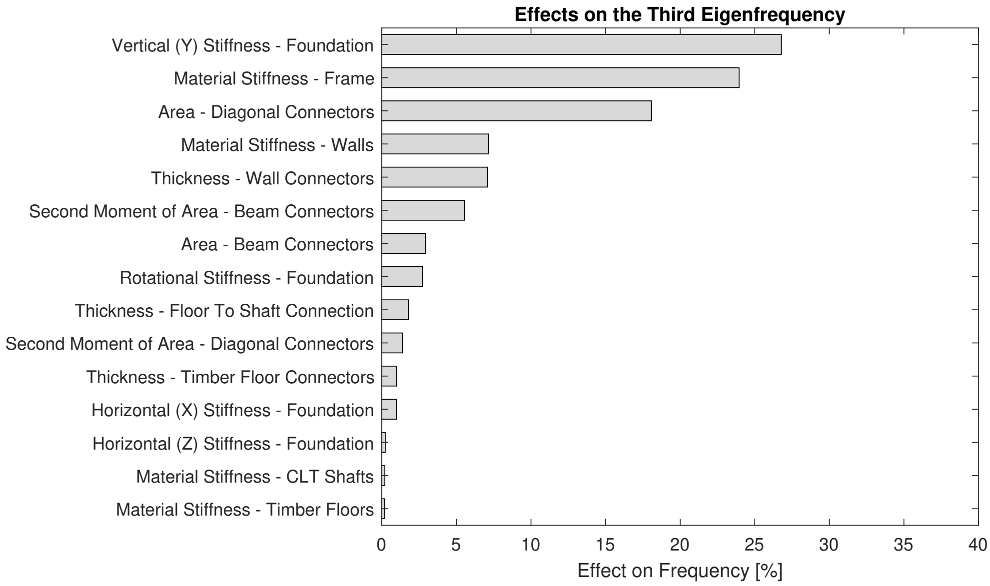


Figure 5.18: Most important parameters for the third mode

5.13 Material Stiffness - Concrete Floors

Since the effect of changing the material stiffness of the timber floors was negligible, a separate test of the influence of the material stiffness in the concrete floors was conducted. The purpose of doing this, is to see if the stiffer concrete floors are governing the stiffness contribution from the floors. This study is done separately from the other parameter studies, and is therefore not a part of the comparison in section 5.12. As the concrete floors have been modelled as an isotropic material, only one modulus of elasticity was altered during the tests. The material stiffness is adjusted from 70% to 130% of the mean stiffness during the tests.

Figure 5.19 show the results from the study. It can be seen that the stiffness of the concrete floors are of a higher importance compared to the stiffness of the timber floors. A possible reason for this is that the higher stiffness in the concrete floors dominates the contribution from the floors. It is likely that a building with only timber floors, will be more influenced by stiffness variations in the floors. Even though variations in the concrete floors have a bigger influence on the fundamental frequencies of Mjøstårnet compared to the timber floors, it is of little importance compared to many of the other parameters that have been studied.

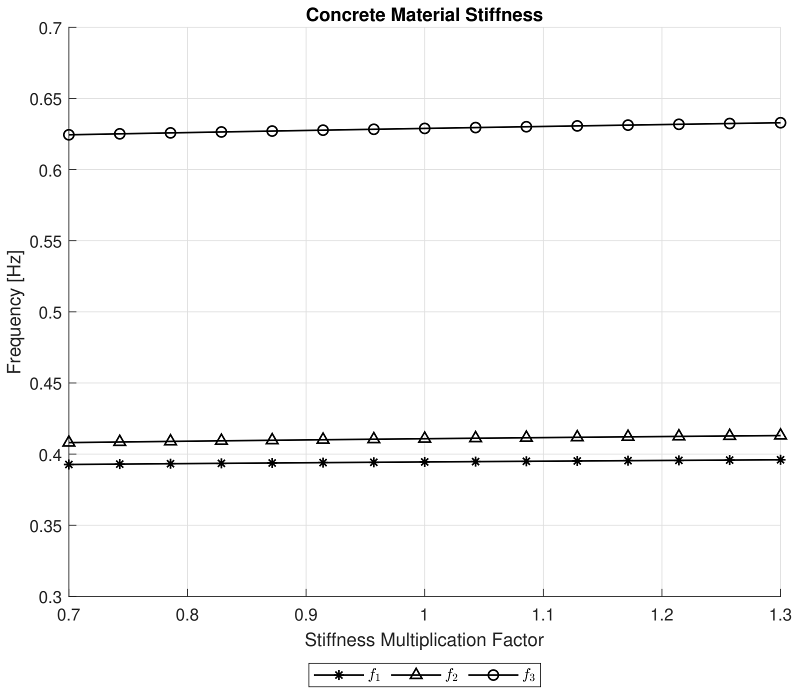


Figure 5.19: The three first eigenfrequencies as a function of concrete floor material stiffness

Chapter 6

Model Updating

A handful of the most significant parameters are selected based on the results from the sensitivity study. The parameters are then updated iteratively to find the values that makes the model able to recreate the behavior of the real life building as accurately as possible. The model described in chapter 4 are used as the starting point of the updating.

A simple model updating routine was programmed in Simulia Isight. The routine makes use of the "Target Solver" block, in combination with the Excel and Simcode components included in Isight. The setup is shown in Figure 6.1.

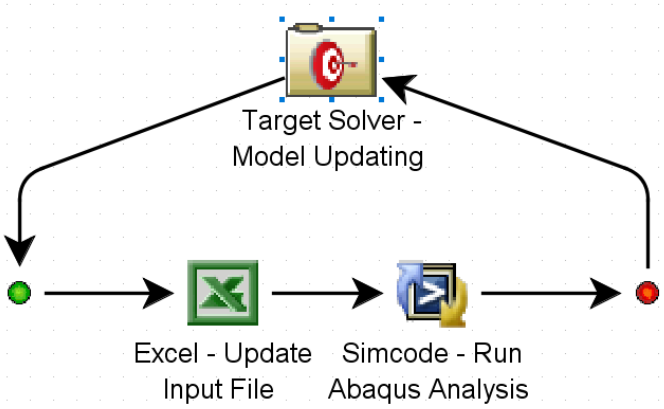


Figure 6.1: Isight setup for model updating

6.1 Input Parameters

A series of runs with slightly different parameters and intervals have been performed, three of which are presented below.

6.1.1 Run 1

The parameters updated in the model updating process for the first run are listed in Table 6.1. The parameters chosen are all related to the stiffness of the structure, while the mass are assumed to accurately modelled. The mass is often considered less uncertain than the stiffness of a structure. One change from the base-model used for the sensitivity study is applied; the uniformly distributed mass (calculated from the imposed loads given in Eurocode 1 part 1-1 [45]) related to the different categories of use (e.g. office, hotel, residential area) are reduced to 50% of the quasi-permanent combination. This change was made because the quasi-permanent load was considered unreasonably high.

Table 6.1: Parameters included in model updating - Run 1

Parameter	Initial Value	Range
Vertical Foundation Stiffness [N/m]	1×10^9	$1 \times 10^8 - 2 \times 10^9$
Material Stiffness - Frame$^{(1)}$	1.0	$0.8 - 1.2$
Material Stiffness - Walls$^{(1)}$	1.0	$0.5 - 2.0$
Diagonal Connector Segments - Area$^{(2)}$	0.2	$0.05 - 0.9$
Beam Connector Segments - 2nd Mom. of Area$^{(2)}$	0.2	$0.05 - 0.9$
Wall Connector Zones - Thickness$^{(2)}$	0.1	$0.075 - 0.8$

$^{(1)}$: Factor multiplied with the mean Young's moduli of the material.

$^{(2)}$: Factor multiplied with the area/thickness/second moment of area of the original beam/shell.

Note that range of allowed values for some of the parameters are more restrictive than the intervals used in the sensitivity study. The decision to restrict some parameters further is taken on the basis on what values are considered probable in real life, for instance it is considered more or less impossible that the connections retain 100% of the stiffness of the member it connects.

6.1.2 Run 2

The parameters picked for the second run (Table 6.2) are almost identical, apart from that the first run only included stiffness parameters, while the second included two mass parameters as well. Hence, the assumption that the mass is accurate do no longer apply. The material stiffness of the frame was also removed as a parameter for the second run, this choice is reasoned with the fact that the large glulam cross sections used in the frame of Mjøstårnet contains so many different lamellae that the stiffness most likely is very close to the mean value.

Table 6.2: Parameters included in model updating - Run 2

Parameter	**Initial Value**	**Range**
Vertical Foundation Stiffness [N/m]	1×10^9	$1 \times 10^8 - 2 \times 10^9$
Material Stiffness - Walls$^{(1)}$	1.0	$0.5 - 2.0$
Diagonal Connector Segments - Area$^{(2)}$	0.2	$0.05 - 1.0$
Beam Connector Segments - 2$^{\text{nd}}$ Mom. of Area$^{(2)}$	0.2	$0.05 - 1.0$
Wall Connector Zones - Thickness$^{(2)}$	0.1	$0.075 - 0.8$
Non-structural Mass - Distributed$^{(3)}$	0.5	$0.25 - 1.0$
Material Density - Walls [kg/m^3]	250	$125 - 375$

$^{(1)}$: Factor multiplied with the mean Young's moduli of the material.

$^{(2)}$: Factor multiplied with the area/thickness/second moment of area of the original beam/shell.

$^{(3)}$: Factor multiplied with the quasi-permanent mass equivalent to the imposed loads.

Note that for the target solver to be able to reach a solution within the specified tolerance, it was necessary to increase the upper bounds for the cross sectional area and the 2$^{\text{nd}}$ moment of area of the connector segments of the beams and diagonals respectively.

6.1.3 Run 3

The model updating parameters chosen for the final model updating are identical to that of the second. However, the upper limit of intervals for the area and 2$^{\text{nd}}$ moment of area of the connections are reduced to 0.9 like in the first run, and the lower limit of multiplication factor for the distributed mass are reduced from 0.25 to 0.20. The parameters along with the specified limits are listed in Table 6.3. The main difference from the previous runs is that the tolerance of the output parameters is increased from 0.001 to 0.0049 (ref. section 6.2).

Table 6.3: Parameters included in model updating - Run 3

Parameter	Initial Value	Range
Vertical Foundation Stiffness [N/m]	1×10^9	$1 \times 10^8 - 2 \times 10^9$
Material Stiffness - Walls$^{(1)}$	1.0	$0.5 - 2.0$
Diagonal Connector Segments - Area$^{(2)}$	0.2	$0.05 - 0.9$
Beam Connector Segments - 2$^{\text{nd}}$ Mom. of Area$^{(2)}$	0.2	$0.05 - 0.9$
Wall Connector Zones - Thickness$^{(2)}$	0.1	$0.075 - 0.8$
Non-structural Mass - Distributed$^{(3)}$	0.5	$0.2 - 1.0$
Material Density - Walls [kg/m^3]	250	$125 - 375$

$^{(1)}$: Factor multiplied with the mean Young's moduli of the material.

$^{(2)}$: Factor multiplied with the area/thickness/second moment of area of the original beam/shell.

$^{(3)}$: Factor multiplied with the quasi-permanent mass equivalent to the imposed loads.

6.2 Output Parameters

The frequencies of the three first modes are chosen as the output parameters. The goal of the model updating is to minimize the difference between the model output and the frequencies measured by Tulebekova et al. [33]. The same targets are used for all three runs, while the tolerance is the same for the first two runs and increased for the third.

Table 6.4: Initial output parameters

Parameter	Initial Model Output	Measured Output$^{(1)}$	Tolerance
f_1	0.422 Hz	0.50 Hz	0.001/0.0049
f_2	0.440 Hz	0.54 Hz	0.001/0.0049
f_3	0.670 Hz	0.82 Hz	0.001/0.0049

$^{(1)}$: From Tulebekova et al. [33]

An important consideration is that multiple combinations of the input variables can cause the same desired output, especially when the amount of output variables are as few as in this case. A extensive test of "Mjøstårnet" involving a "shaker" and detailed instrumentation is planned as a part of the DynaTTB project [2]. The test will provide measured frequencies for many more modes, as well as information about mode shapes, damping properties etc. that can be added to the list of targets for the model updating and improve the accuracy and certainty of the results considerably. With more output/target variables the number of input variables may also be increased. The model updating in this thesis is therefore intended to be seen more as a demonstration of the method and an estimate rather than a strict answer to the values of the input variables.

6.3 Results

6.3.1 Run 1

A total of 35 iterations were needed for the target solver to find a solution within the specified tolerance of 0.001. The value of the input parameters before and after the first run of the updating are presented in Table 6.5, and the convergence of the frequencies is plotted in Figure 6.2. It is clear from the results that the stiffness of the initial model was underestimated, since the general trend is that the value of the parameters related to stiffness are increased. Another possibility is that the mass is overestimated in the initial model, however the mass of a structure is often considered a more certain quantity than the stiffness.

Table 6.5: Initial and updated input parameters - Run 1

Parameter	Initial Value	Final Value
Vertical Foundation Stiffness [N/m]	1×10^9	1.238×10^9
Material Stiffness - Frame$^{(1)}$	1.0	1.168
Material Stiffness - Walls$^{(1)}$	1.0	1.0
Diagonal Connector Segments - Area$^{(2)}$	0.2	0.9
Beam Connector Segments - 2 Mom. of Area$^{(2)}$	0.2	0.9
Wall Connector Zones - Thickness$^{(2)}$	0.1	0.583

$^{(1)}$: Factor multiplied with the mean Young's moduli of the material.

$^{(2)}$: Factor multiplied with the area/thickness/second moment of area of the original beam/shell.

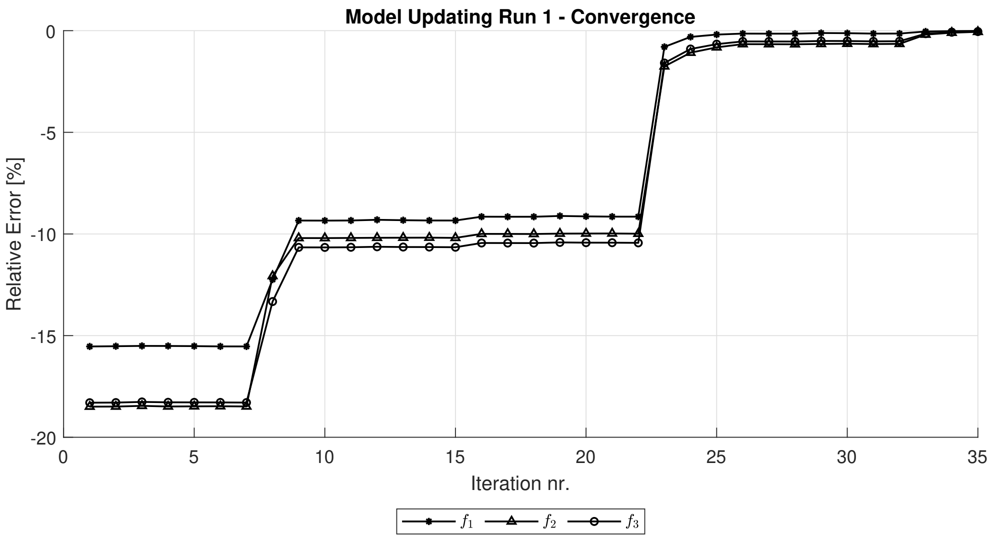


Figure 6.2: Convergence of model updating procedure - run 1

6.3.2 Run 2

For the second run of the model updating it was necessary to increase the upper limit for the area and 2nd moment of inertia of the connector segments for the target solver to be able to reach a solution within the specified tolerance of 0.001. After 58 iterations the solution presented in Table 6.6 was found. The results shows the same tendency as in the first run; the stiffness in the initial model is underestimated. This conclusion remains after the imposed mass is included as a parameter in the updating and is reduced during the process to a final value that is approx. 50% compared to the (fixed value) mass applied in the first run.

Table 6.6: Initial and updated input parameters - Run 2

Parameter	Initial Value	Final Value
Vertical Foundation Stiffness [N/m]	1×10^9	2×10^9
Material Stiffness - Walls$^{(1)}$	1.0	1.307
Diagonal Connector Segments - Area$^{(2)}$	0.2	1.0
Beam Connector Segments - 2nd Mom. of Area$^{(2)}$	0.2	1.0
Wall Connector Zones - Thickness$^{(2)}$	0.1	0.8
Non-structural Mass - Distributed$^{(3)}$	0.5	0.252
Material Density - Walls [kg/m^3]	250	320.3

$^{(1)}$: Factor multiplied with the mean Young's moduli of the material.

$^{(2)}$: Factor multiplied with the area/thickness/second moment of area of the original beam/shell.

$^{(3)}$: Factor multiplied with the quasi-permanent mass equivalent to the imposed loads.

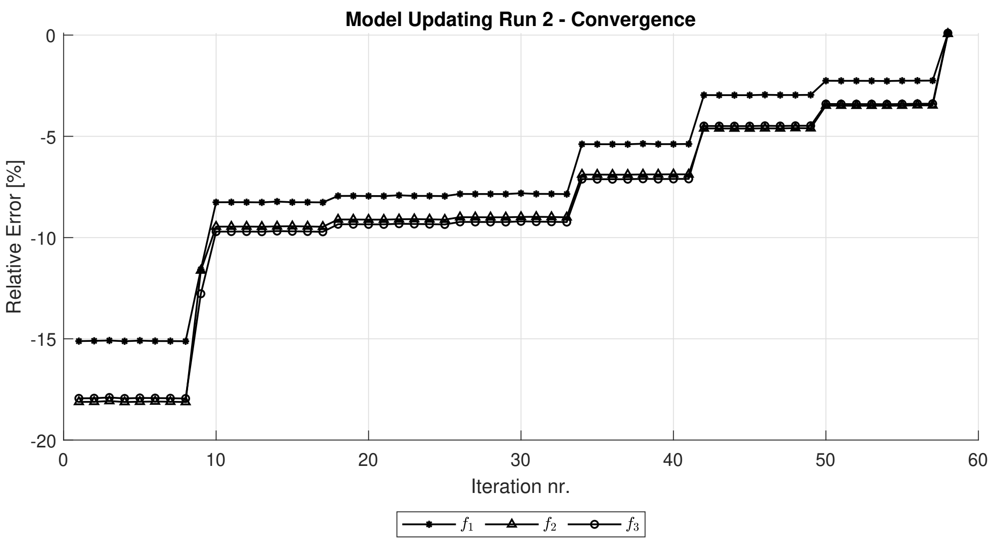


Figure 6.3: Convergence of model updating procedure - run 2

6.3.3 Run 3

When the tolerance of output parameters was increased and the lower limit of the distributed load multiplication factor was decreased slightly compared to the previous runs, the target solver managed to find a solution without needing 100% connector stiffness. A total of 50 iterations were needed to achieve the solution in Table 6.7.

Table 6.7: Initial and updated input parameters - Run 3

Parameter	Initial Value	Final Value
Vertical Foundation Stiffness [N/m]	1×10^9	2×10^9
Material Stiffness - Walls$^{(1)}$	1.0	1.310
Diagonal Connector Segments - Area$^{(2)}$	0.2	0.9
Beam Connector Segments - 2nd Mom. of Area$^{(2)}$	0.2	0.9
Wall Connector Zones - Thickness$^{(2)}$	0.1	0.8
Non-structural Mass - Distributed$^{(3)}$	0.5	0.206
Material Density - Walls [kg/m^3]	250	313.1

$^{(1)}$: Factor multiplied with the mean Young's moduli of the material.

$^{(2)}$: Factor multiplied with the area/thickness/second moment of area of the original beam/shell.

$^{(3)}$: Factor multiplied with the quasi-permanent mass equivalent to the imposed loads.

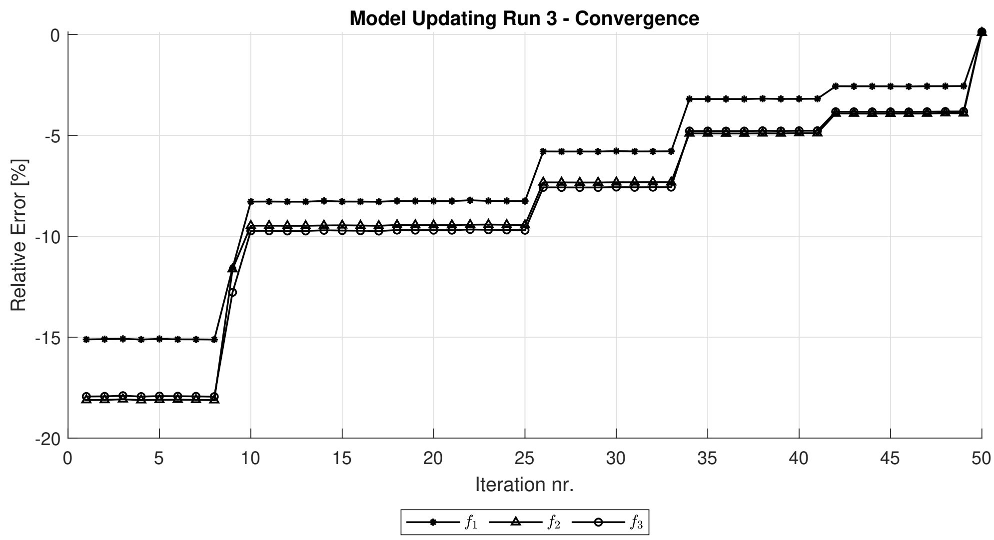


Figure 6.4: Convergence of model updating procedure - run 3

6.4 Summary

From the three model updating runs some general observations can be made. All three runs show the same tendency in that the stiffness of the base model is underestimated. For instance, the connection parameters for the frame is set to the upper limit for the best fit in all runs. Also, the foundation stiffness is significantly increased for all runs. The stiffness of the exterior walls, both in terms of material stiffness and connection-zone thickness, is generally increased.

The two runs that include mass as parameter, also show the same tendencies: non-structural loads on floors are overestimated in base model and the material density of outer walls is underestimated. This might be a real effect, but it can also be due to the randomness of the Isight procedure.

Although the tendencies are the same, the output values from all three runs differs. This is especially evident for the values of the stiffness parameters. The added stiffness is distributed differently in the model in all three runs. Two important notices can be made from this:

- In order to obtain good and reliable predictions of parameter values from a model updating procedure, it is important to base it on a larger amount of target values. In this case, it can be in the form of more frequencies. Including mode shapes as targets is also likely to improve the reliability of the estimations significantly.
- The input of the base model, and the allowed range for the parameters in model updating procedure, should be based on values that are feasible. This will reduce the chance of obtaining a combination of parameters that does not represent the real structure.

Chapter 7

Wind Loads

This chapter is meant to be a demonstration of some of the possibilities for doing wind-related analyses using the scripts that are developed and included in the digital appendix of this thesis. The basis of the calculations is *Eurocode 1: Actions on structures - Part 1-4: General actions - Wind actions* [23] and its Norwegian national annex. A detailed explanation on the theory behind wind actions, as well as a review of the calculation procedure, including the most important equations, are given in section 2.3 of the background chapter.

The model used in the analysis is set up to resemble Mjøstårnet with the input described in chapter 4, with the values of some of the parameters improved by the model updating performed in chapter 6. The results of the final (subsection 6.3.3) of the three model updating runs are used. However as mentioned previously, the main focus of this thesis as been the development of the parametric model, not to get all the input exact for Mjøstårnet. As a consequence the results presented the results section of this chapter must be seen as demonstration of the capabilities of the script and an estimate of the values to be expected for Mjøstårnet, rather than an accurate solution.

7.1 Estimation of Parameters

A modal dynamics step is implemented in the wind analysis procedure. The step is designed to model a free vibration time history, initialized by an impulse load. The free vibration time history is then used to determine some basic parameters for use in the upcoming wind-related calculations.

7.1.1 Frequency

The frequency of a system can be determined by measuring the the time it takes for the system to complete one or more cycles of vibration. The equation for the frequency f as a function of the period $T = t_{i+1} - t_i$, where t_i is the elapsed time at the i^{th} peak, is:

$$f = \frac{1}{T} \tag{7.1}$$

or averaged over n cycles:

$$f = \frac{n}{t_{i+n} - t_i} \tag{7.2}$$

The default option implemented in the script is to measure the frequency over $n = 2$ peaks, starting from the second peak. The reason for not measuring from the first peak is to minimize the risk of the frequency being influenced by the impulse load used to initialize to free vibration. Figure 7.1 shows a "XYPlot" from Abaqus with the time span illustrated.

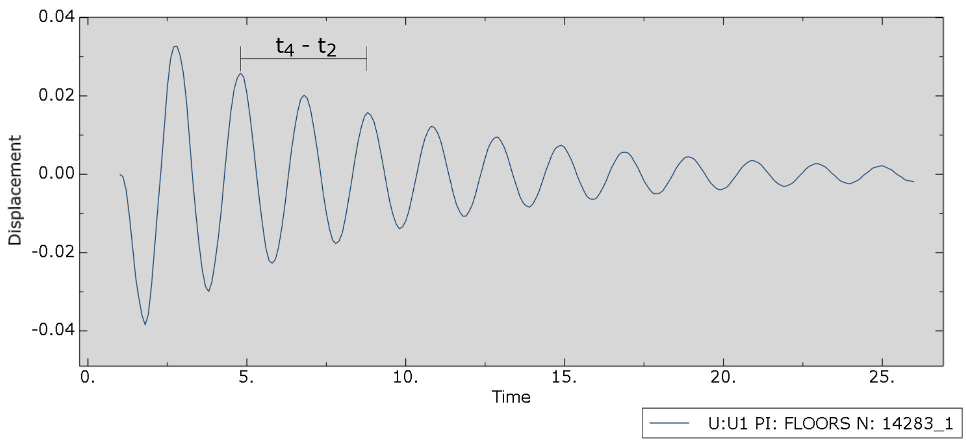


Figure 7.1: Estimation of natural frequency based on free vibration

7.1.2 Damping Values

The damping ratio for a mode can be estimated by the method of logarithmic decrement. The method is accurate for lightly damped structures, which is the case for most civil engineering structures. The logarithmic decrement, δ, is found by taking the natural logarithm of the ratio between the magnitudes of two subsequent peaks:

$$\delta = \ln \frac{x_i}{x_{i+1}} \tag{7.3}$$

If the damping is independent of the magnitude of the deformations, the accuracy of the measurements can be improved by averaging over n cycles [47]:

$$\delta = \frac{1}{n} \ln \frac{x_i}{x_{i+n}} \tag{7.4}$$

It can be shown that the damping ratio ζ can be calculated from the logarithmic decrement using Equation 7.5:

$$\zeta = \frac{1}{\sqrt{1 + (\frac{2\pi}{\delta})^2}} \tag{7.5}$$

Alternatively when $\zeta \ll 1.0$:

$$\zeta \approx \frac{\delta}{2\pi} \tag{7.6}$$

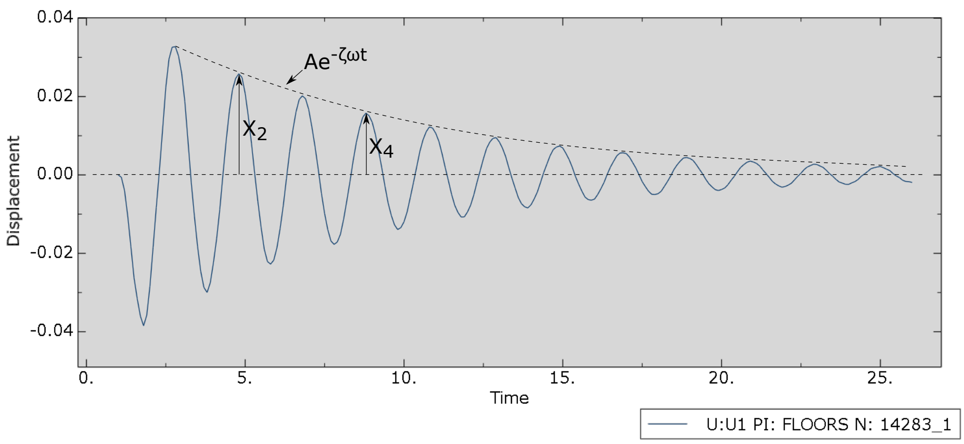


Figure 7.2: Estimation of logarithmic decrement based on free vibration

The plot in Figure 7.2 shows the peaks ($i = 2$, $n = 2$) used for estimation of damping in the script, as well as the envelope curve defined by the exponential expression $x(t) = Ae^{-\zeta \omega t}$.

7.2 Method

The main focus of the analyses performed in this section are the output related to the accelerations of the building. A parameter study has been performed, where selected parameters related to the damping of the structure and the wind conditions at the site have been modified to study their effect on the resulting acceleration response.

The main script used for running the analyses is *TTB_3D_EC_Wind.py* which is designed for the exact purpose of analysing the response of a parametric structure to wind actions. To be able to execute many iterations with different parameters efficiently, an Isigth routine similar to the one used for the sensitivity study in chapter 5 is used. Instructions for setting up a parameter study like the one shown in Figure 7.3 are given in section A.4 of the user guide included in the appendix.

Almost all the analyses in this chapter are performed with the wind coming from the x-direction (i.e. perpendicular to longest side of the building), however the comparison with the threshold values in section 7.5 includes both horizontal directions.

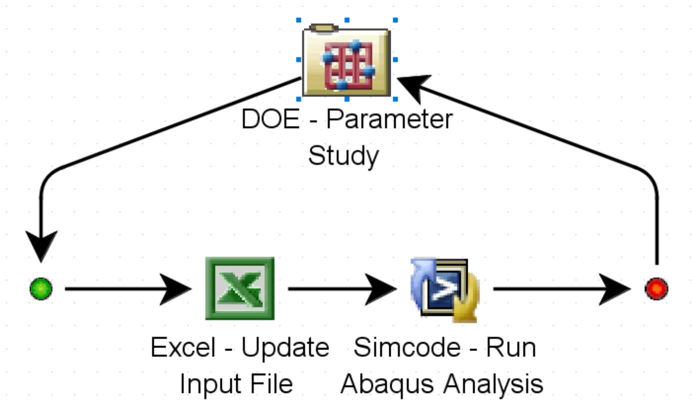


Figure 7.3: Isight setup for a parameter study

7.3 Verification of Calculations

Before presenting and interpreting the acceleration results, some simple test are performed to verify that the script functions as intended.

7.3.1 Damping Measured in the Free Vibration Analysis Step

Since the damping parameters used in the wind calculations are based off the results of a analysis step simulating free vibration, it was necessary to check that the measured damping in the free vibration step corresponds well with the applied damping. The test was performed by applying a direct modal (global) damping to the first 10 modes of the system. The mode excited in the free vibration step is similar to the first mode of the system, hence the damping measured in the free vibration step should be similar to the applied modal damping. The results are plotted in Figure 7.4, with the applied and measured damping along the x- and y-axis respectively.

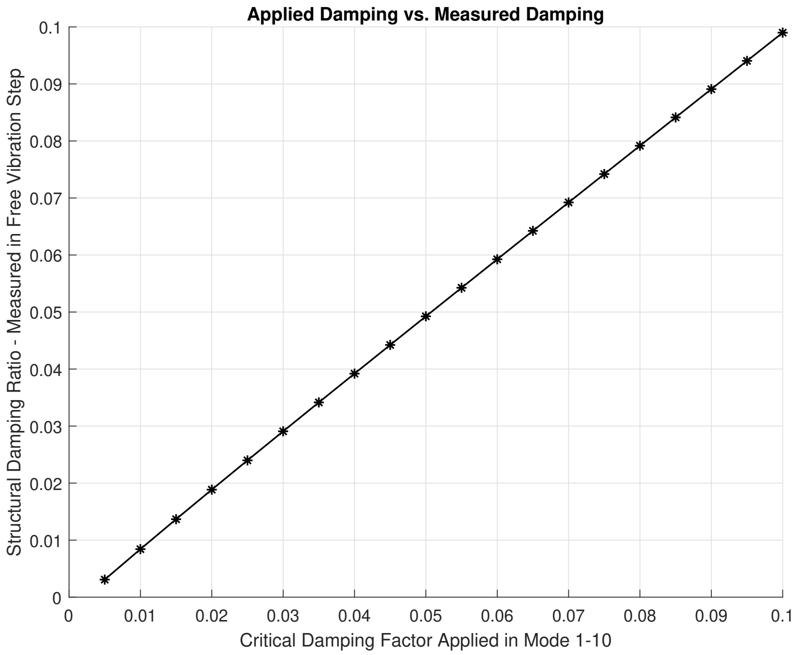


Figure 7.4: Applied damping versus damping measured in free vibration step

The results shows good correlation between the applied and measured damping, maybe with a trend of the measured damping being slightly underestimated.

7.3.2 Frequency Measured in the Free Vibration Analysis Step

The first natural frequency in the wind direction used in the wind calculations is also determined (measured) on the basis of the same free vibration analysis step as the damping. By including a frequency step for determining the natural frequencies based on the eigenvalues of the system, the two frequencies can be compared and the accuracy of the "measured" frequency can be assessed.

	Free Vibration Step	Frequency Step
f_1	0.500 Hz	0.496 Hz

Table 7.1: First frequency in the wind (x-) direction

The results in Table 7.1 shows less than 1% deviation between the two different ways of calculating the frequency, which is considered to be well within the acceptable margin of error.

Note that the accuracy of both the measured damping ratio and frequency should be expected to decrease if the mode investigated is a mode that consists of a combination of e.g. translation and torsion, or translation in more than one of the global axes. This is because the parameters are determined based on the displacements in the wind direction, sampled at the centre of the top floor. It is always advised to verify that the results from the free vibration step seems reasonable. One way to check the results is to create an "XYPlot" in Abaqus and check that at least the first few cycles are like a sinusoidal, without any additional local peaks. Another way to get a quick indication of the quality is to check that the frequency calculated in the free vibration step matches the frequency of the corresponding mode in the frequency step. If the deviation between the frequencies are small, one can be fairly confident that the script were able to identify the peaks of the cycles correctly, hence the damping estimate should be of decent quality as well.

7.4 Results - Acceleration

The results of the different analyses performed are presented in the following sections.

7.4.1 Structural Vs. Aerodynamic Damping

The damping used in the Eurocode for wind-related calculations are the sum of the damping in the structure and the aerodynamic damping. The structural damping are determined by modelling free vibration in Abaqus, while the aerodynamic part are determined by equation F.16/F.17 in the Eurocode. Note that the Eurocode measures damping as logarithmic decrement, however the values are converted to a damping ratio for the purpose of this thesis. The structural, aerodynamic and total damping are presented in the figure below.

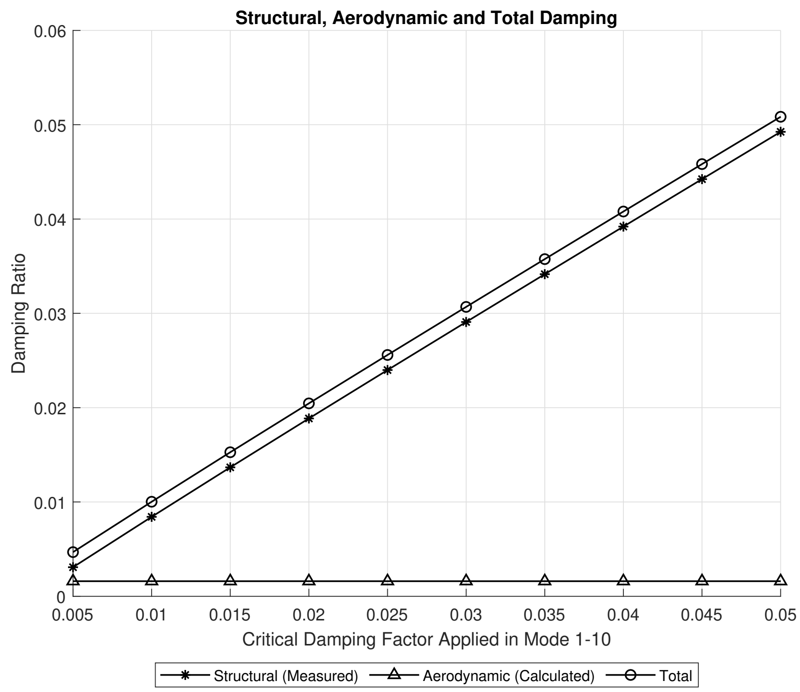


Figure 7.5: Structural, aerodynamic and total damping

As shown in Figure 7.5 the aerodynamic damping ratio is independent of the structural damping. The aerodynamic damping ratio is in this case around 1% and can be an important contribution to the total damping of the system, especially if the structural damping is low.

7.4.2 Peak Acceleration

The peak value is the highest acceleration that is expected to occur within a specified return period. Guidelines/threshold values for peak acceleration of many different structures are given in ISO:10137 [25], based on a return period of 1 year.

With the one year return period, the resulting peak acceleration at the top floor (excluding the rooftop terrace) of the building are plotted against the damping applied directly in mode 1-10 of the model in Figure 7.6.

The results shows that increasing the damping of a structure reduces the accelerations significantly. For instance a damping ratio of 2.5%, which is not unrealistic for a timber structure, shows less than half the acceleration compared to structure with 0.5% applied modal damping.

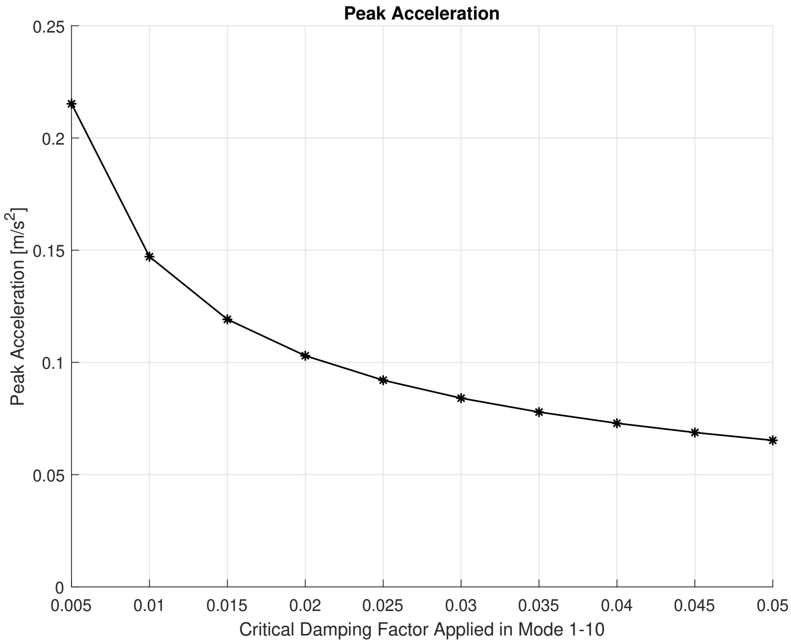


Figure 7.6: Peak acceleration at highest level below rooftop terrace

7.4.3 Standard Deviation of Acceleration

Some prefer to state the acceleration as a standard deviation instead of giving the peak value. The standard deviation is also commonly denoted RMS or *root mean square* in the literature. The peak acceleration is defined as the standard deviation multiplied by a peak factor k_p which is simply a function of the natural frequency. Hence, it is only a matter of preference, or the units of the threshold values in the codes/guidelines, whether the results are stated as a standard deviation or as a peak value. For instance ISO:6897 [48] specifies threshold values in terms of the standard deviation.

The standard deviation of the acceleration as a function of the applied modal damping is plotted in Figure 7.7.

Note that the results for the standard deviation is the same as the peak acceleration reduced by the peak factor k_p, which is constant for all relevant damping values ($\zeta \ll 1$, such that $\omega_d \approx \omega_n$). This confirms what is stated above; that in practice it does not matter if the acceleration is given as a peak value or standard deviation. The peak factor of the structure investigated is found to be approx. 3.56.

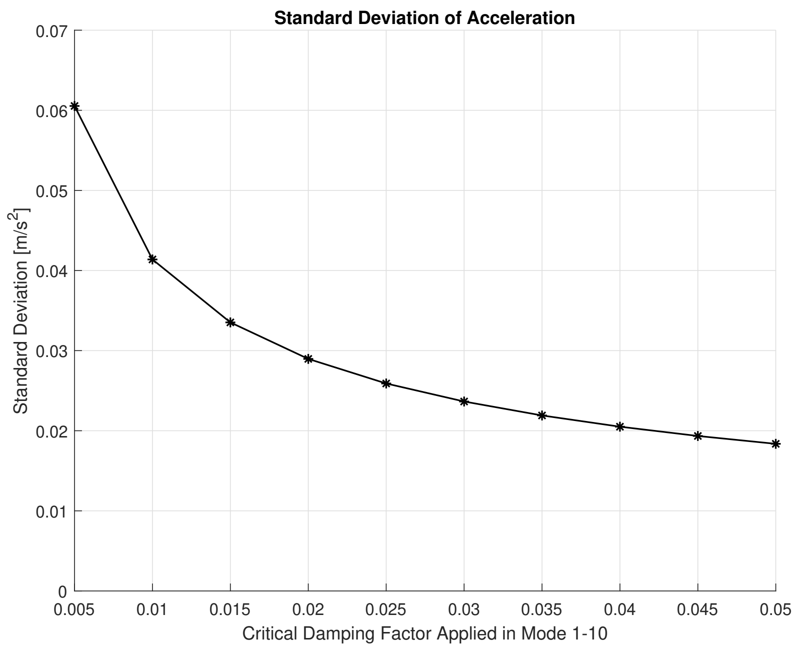


Figure 7.7: Standard deviation of the acceleration at highest level below rooftop terrace

7.4.4 Acceleration at Different Levels

The acceleration varies at the different levels of the building, and it is usually only at the top few floors where too large accelerations is a problem. The parametric model allows the user to specify at what height the acceleration should be calculated, hence the accelerations can easily be determined for every level of the building. The results of this feature is demonstrated below.

Before running the analysis the damping values identified by Tulebekova et al. [33] listed in Table 7.2, are added to the model. The rest of the model is the same resulting model from the third run of the model updating in chapter 6 as used previously. The acceleration response are shown in Figure 7.8 for level 0 (ground level) to level 17 (rooftop terrace).

Table 7.2: Damping values from Tulebekova et al. [33] (Mean values from 6 DD-SSI analyses (March 2019 - May 2019))

Mode	Applied Damping
1	1.685%
2	2.458%
3	1.863%

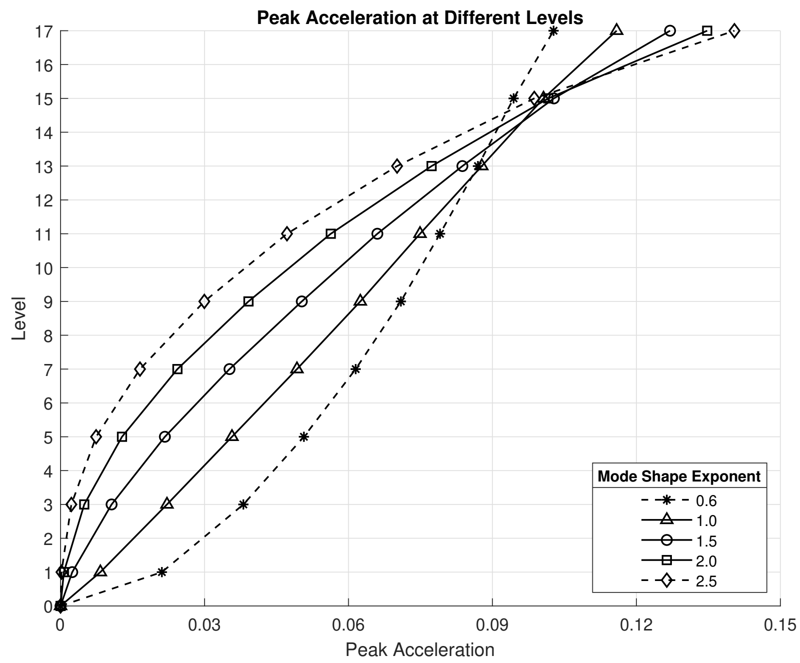


Figure 7.8: Peak acceleration at different levels for different mode shapes

The different lines in Figure 7.8 each represent different values of the exponent, ζ, used to approximate the first mode shape in accordance with equation F.13 in the Eurocode [23]:

$$\Phi_1(y) = \left(\frac{y}{h}\right)^{\zeta} \tag{7.7}$$

Note that the variable z is replaced with y in Equation 7.7 to match the orientation of the axes used in the model, and that ζ in this expression is not related to the damping ratio, also denoted ζ. Appendix F to the Eurocode [23] recommends the following choice of ζ:

Table 7.3: Recommended mode shape exponents (from [23])

ζ	**Building Type**
0.6	Slender frame structures with non load-sharing walling or cladding.
1.0	Buildings with a central core plus peripheral columns or larger columns plus shear walls.
1.5	Slender cantilever building and buildings supported by central reinforced concrete cores.
2.0	Towers and chimneys.
2.5	Lattice steel towers.

Based on the recommendations shown in Table 7.3, a mode shape exponent of $\zeta = 1.5$ are used for the remainder of the analyses. However, the results presented in this section shows that getting the mode shape right is important for the acceleration estimates to be accurate. Both the max acceleration at the top of the structure, as well as the distribution in the lower parts of the structure are heavily influenced by the chosen exponent.

7.4.5 Acceleration at Different Return Periods

The return period indicates the probability for a acceleration value (or any other quantity) to be exceeded a given year, refer to section 2.3.3 for further explanation of the relation between annual exceedance probability and return period. As an uncomplicated, but slightly inaccurate explanation one can say that a value with e.g. a return period 50 years is estimated to be exceeded once every 50 years on average.

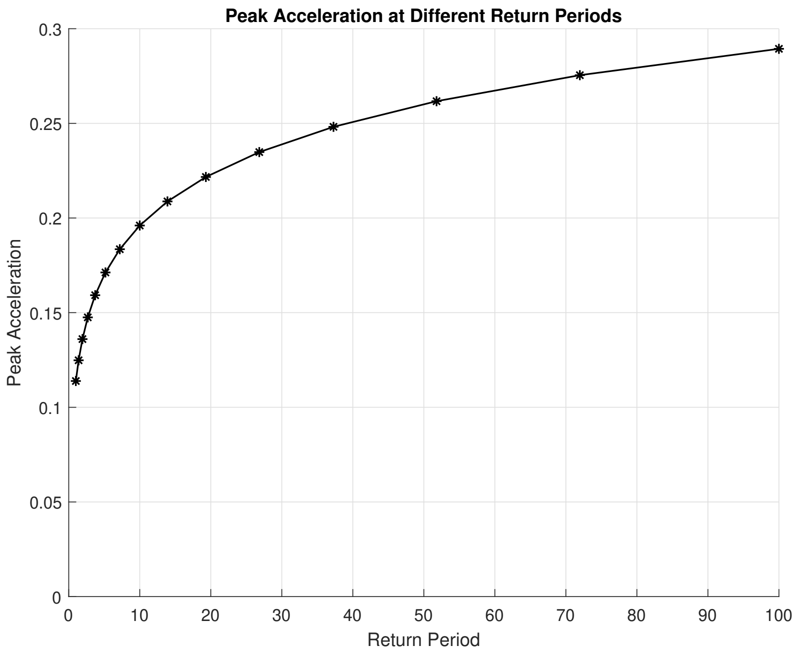


Figure 7.9: Peak Acceleration at different return periods

Figure 7.9 shows the results peak acceleration value at the highest level below the rooftop terrace, plotted against return periods of the wind actions ranging from 1 to 100 years. For serviceability design it is common to use short return periods, e.g. one or five years [25][48], since the consequences of exceeding a threshold value are small. For ultimate limit state design, however, longer return periods are used (typically 50 years [23]), and much higher actions/response of the structure is expected. It should be noted that the acceleration is rarely a ULS design problem, however increasing the return period of the wind action will also increase the forces/stresses in the structure, in a similar manner as for the accelerations.

7.4.6 Accelerations at Different Wind Speeds

As the final parameter the effect of the wind velocity is investigated. Note that the values along the x-axis are the mean wind velocity v_m at the reference height for the structural factor $z_s = 0.6h$, and not the fundamental value of the basic wind velocity $v_{b,0}$. The values are almost the same; the mean value are the basic wind velocity multiplied with factors adjusting for e.g. the terrain and altitude at the site. Hence, the mean value is the most accurate to use for comparison with on-site wind measurements.

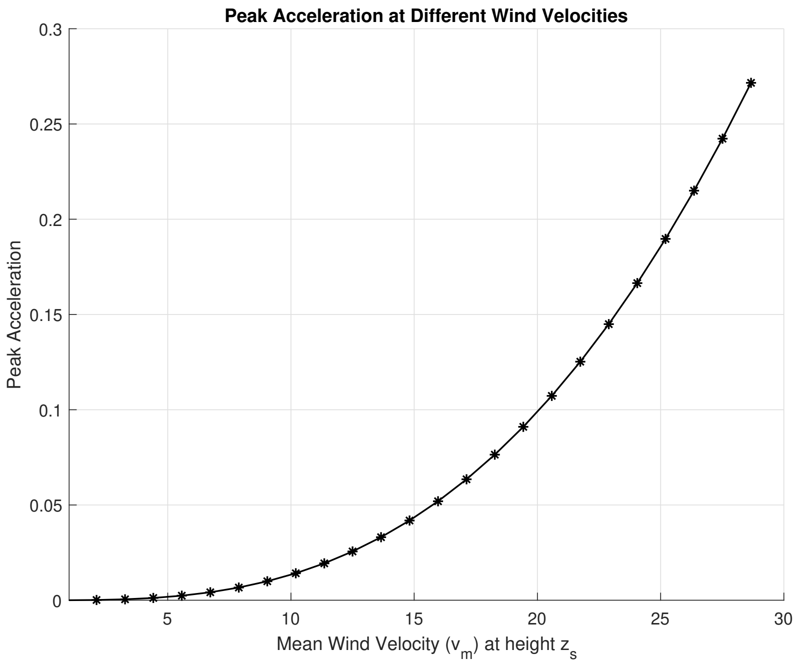


Figure 7.10: Peak Acceleration at different mean wind velocities

The peak acceleration grows exponentially as the mean wind velocity increases (Figure 7.10). This response is as expected, due to the velocity pressure being a function of, amongst other, the mean wind velocity squared, recall Equation 2.31 of the background chapter:

$$q_p(z) = \frac{1}{2} \cdot \rho \cdot v_m^2(z) \cdot [1 + 2k_p I_v(z)] \tag{2.31}$$

Tulebekova et al. [33] have measured the accelerations and the wind velocity of Mjøstårnet. Their results are plotted along with the results from the parametric model in Figure 7.11:

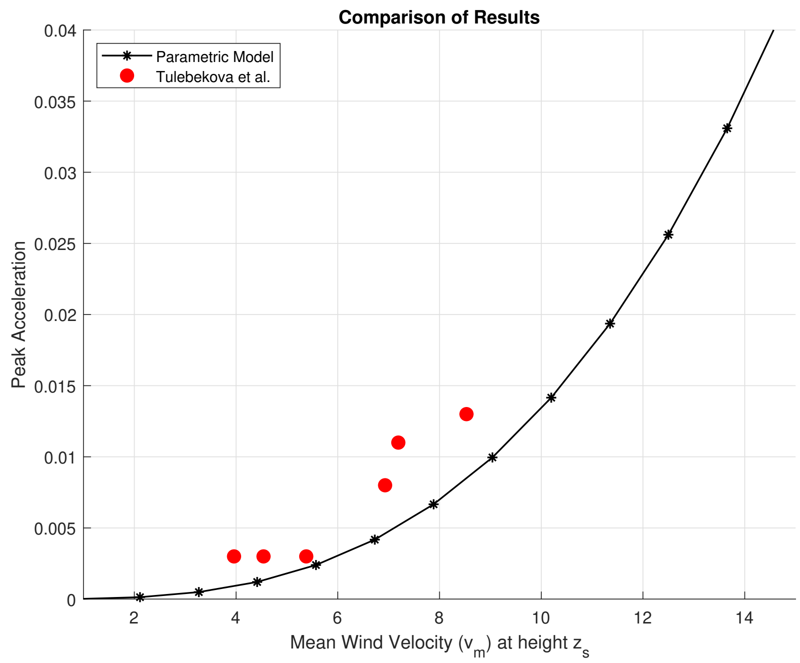


Figure 7.11: Comparison of results

The results shows that the results from Tulebekova et al. gives slightly higher acceleration values than the parametric model. However the fact that the sensors used in the experiments are placed a little higher on the structure (ref. subsection 2.5.3), than the level where acceleration are sampled in the parametric model (highest floor below to rooftop terrace), may contribute to make the actual deviation less than it appears to be in the plot.

7.5 Comparison with ISO10137 Guidelines

Peoples perception of vibration is highly subjective. An acceleration level that causes discomfort or even motion-sickness symptoms for one person, may be barely noticeable for other people. The perception of acceleration is also highly dependant on the frequency of the vibrations [21]. ISO10137 [25] provides recommendations for serviceability design of buildings and walkways. The recommended threshold values for horizontal motion are presented in Figure 7.12, the upper line (1) marks the threshold for offices, while the lower (2) is for the design of residential buildings. The limit for residential buildings are set to when the perception probability is approximately 90% [25].

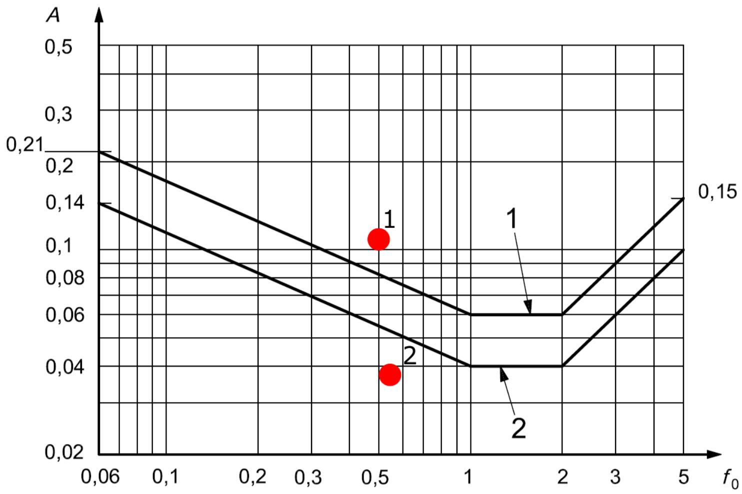


Figure 7.12: Threshold values for vibrations in buildings (Figure D.1 from ([25])

The red dots added to the plot marks the peak acceleration values in both directions acquired using the updated base setup of the parametric model, with the damping values listed in Table 7.2. The results are also listed in Table 7.4.

Table 7.4: Peak acceleration response for wind in x- and z-directions

Marker nr.	Direction	Peak Acceleration
1	X (Transversal)	0.114
2	Z (Longitudinal)	0.038

The results indicates that the peak acceleration value at the highest non-roof level of Mjøstårnet, in the transverse direction are above the recommend threshold.

7.6 Static Displacement

Although the main focus of this chapter has been on accelerations, the script also calculates the equivalent static load according the the Eurocode [23]. The load is converted to line loads which are applied to the columns of the frame structure, as illustrated in Figure 7.13.

The resulting deformation in the global x-direction from the equivalent static wind load on the updated model of Mjøstårnet, with the damping values listed in Table 7.2, are shown in Figure 7.14.

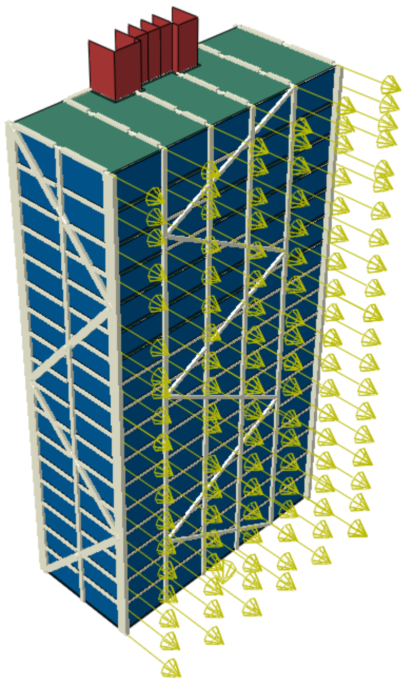


Figure 7.13: Application of static wind load

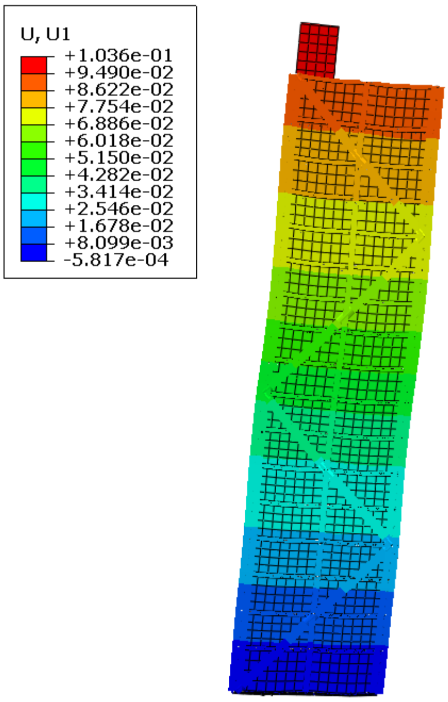


Figure 7.14: Static displacement [m]

Chapter 8

Discussion

The results of each separate test is already presented and discussed in the respective sections. Hence, the purpose of this chapter is to tie the different parts of the thesis together and try to make connections between the different results. The parametric model is also discussed. Both things that have worked as intended, as well as issues and possible sources of error that have been revealed in the process are presented. Finally possible solutions to the issues and ways to increase the accuracy are discussed.

8.1 Results

This section will mainly focus on the observations made in the sensitivity study, as the other analyses were mostly conducted as demonstrations of the capabilities of the script, and based on input that in some cases are rough estimations or even mere guesses.

The results of the sensitivity study can be used to sort out what parameters it is important to make sure are correct in order to create a realistic model. The material stiffness of the frame is one of the most influential parameters for all modes. However, this is also the stiffness parameter that most easily can be predicted with good accuracy. As stated in subsection 6.1.2 in the model updating chapter, the large cross-sections required for the glulam members of tall timber buildings will result in little variation in stiffness throughout the frame. Naturally, some degree of variation will occur due to e.g. variation in moisture content. For this kind of structure, these variations are often limited as the frame usually is sheltered from the weather, resulting in a Service Class 1 structure.

The stiffness of the frame connections, in particular the axial stiffness of diagonal connections, is an important parameter. For the kind of building studied, diagonals are the main contributor to the horizontal stiffness. The result is therefore as expected. The rotational stiffness of beam connections is also a relatively important parameter, although less so than the diagonal connections. Determining the stiffness of connections is not as straight forward as for the members. Eurocode 5 part 1-1 [49] provides a simple way of predicting the stiffness of connections by introducing the slip modulus, K_{ser}, of connections for use in SLS. Studies does however show that the stiffness obtained from this procedure vary significantly from what is measured, and is generally underestimated [50] [51] [52]. Nonetheless, the Eurocode does provide a better initial prediction for the connector stiffness than the guesses used for the base model in this thesis. When the model is paired with more output variables (ref. section 6.2), model updating can hopefully be a useful tool for making good estimates for finding the connector stiffness.

The vertical foundation stiffness of a building is very hard to predict, as the soil conditions will vary from building to building. Regardless of this, the study demonstrates the importance of getting a good prediction of the parameter, as it is the single most important parameter. In fact, if the stiffness of the foundation is very low, other measures for increasing the horizontal stiffness of the structure will become trivial. In this case the building will rotate at its base while the structure it self will act as a rigid body. Tulebekova *et al.* [33] suggests that the prediction of the foundation stiffness of Mjøstårnet done by Sweco underestimates the actual stiffness. This might be the reason for the difference between the calculated and measured frequencies and direction of modes. The results from the sensitivity study supports this suggestion. Simply put, a good estimation of the foundation stiffness is essential in order to be able to predict the dynamic behaviour of a structure.

The final parameters that will be highlighted is the parameters related to the stiffness of the exterior walls, both in terms of the stiffness of the fictitious material used and the stiffness of the connections. Both parameters are shown to be important, especially for the longitudinal bending mode and torsional mode. The possible reason to why they have little influence on the transverse bending mode is that the amount of walls spanning in this direction are much lower compared to the walls spanning in the longitudinal direction, thus introducing less stiffness to transverse bending. The results questions the assumption usually used during design, that the contribution from non-structural external walls towards the horizontal stiffness of a building can be neglected. It is important to consider that the magnitude used for these parameters in the base model are highly uncertain, and it has not been verified if they are realistic. In order to be able to conclude how much of an impact external walls might have on the behaviour of a building, it is necessary to establish better predictions of their stiffness. Regardless, the results found in this thesis are interesting, as it allows for an alternate explanation to why predicted modes of Mjøstårnet have frequencies that are significantly lower and of different direction compared to the measurements of the finished building. It is likely that it can be explained by a combination of the effects from the vertical foundation stiffness being underestimated and the exterior walls not being represented in the model.

Some parameters have very little influence on the dynamic behavior of the model. It is valuable to have an overview of these factors, as a rough estimate of this variables can still produce results of sufficient accuracy. Time and resources can therefore be saved. As an example, a considerable amount of time was spent on finding the parameters to define the section of the timber floor elements. However, the influence of the stiffness in the floors proved to be of very little importance. The parameters found to fall under this category are listed below:

- Horizontal foundation stiffness (as long as it is not too low)
- Rotational stiffness of foundations
- Timber floor material stiffness
- Timber floor connection stiffness
- Material stiffness of shaft walls
- Rotational stiffness of diagonal connections
- Axial stiffness of beam connections

It must be considered that not every parameter of the model have been studied. Some of the parameters where expected to be of little importance, and therefore neglected in the studied. However, it is possible that some of the expectations were wrong, leading to important parameters being undiscovered.

8.2 Parametric Model

By far, the most of the time spent working on this thesis have been dedicated to the development of the parametric model. Resulting in between 6000 and 7000 lines of Python code (excluding many scripts discarded during the process), an extensive excel file for setting up the input and several Isight models, all of which is included in the digital appendix. The model has later been used in several different tests and analyses, including a sensitivity analysis of mainly stiffness parameters (chapter 5), several model updating runs and finally a series of wind-related parameter studies. Several useful experiences were made about the functioning of the model, a selection of things that work well, an addition to some of the problems encountered follows.

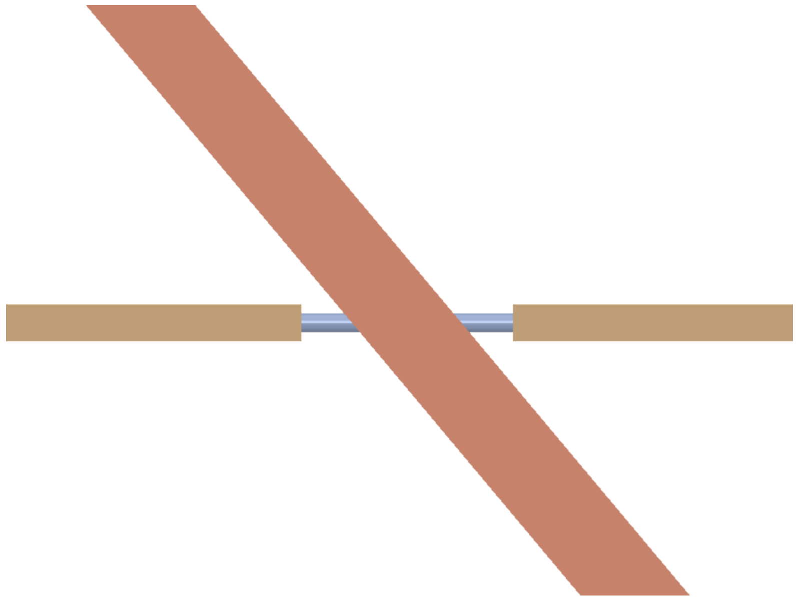


Figure 8.1: Typical connection between two beams and a diagonal

8.2.1 Modelling of Connections in Beam Elements

The method of using connector segments (ref. subsection 3.7.1 and Figure 8.1) to model the reduced stiffness of connections between the different parts of the frame structure proved to be successful. The effect of changing either the area or one of the 2nd moments of areas have a clear effect on the stiffness of the structure. The effects of change in axial stiffness of the connections can most easily be seen in the part of the sensitivity study concerning the diagonals, while the change of rotational stiffness becomes clear when dealing with the horizontal beams of the structure.

There are some doubts related to the validity of using beam theory for the finite element formulation of the short connection. However, for the purpose of the global analyses the model are intended for a lower accuracy of the stiffness in the connection segments are deemed acceptable. The method where the stiffness are controlled by tweaking the cross sectional properties of is in itself not the most accurate way of modelling stiffness in connections, but the fact that it is easy to understand and interpret the input values was important in the choice of method. Another important factor to consider is that the chosen method has low risk of failing. Such that if a connector segment for some reason are not generated the parts will still be connected, and the consequences will be much less severe than missing connector if the parts of the structure was created separately and connected through e.g. springs.

A potential downside of using the connector segments is that they are unfit for stress analysis, and local analysis of the structural members in general. Since the main focus of this thesis are directed towards the global behavior of structures under service loads, this was not considered a problem.

8.2.2 Modelling of Connections in Shell Elements

A similar approach as for the connections in beam elements where chosen for the shell elements, primarily used to model walls and floors. The wall panels are enclosed by a connector zone with reduced stiffness as demonstrated in Figure 8.2. The chosen approach worked well as long as the thickness was not reduced too much. If the thickness becomes too low, local, spurious modes will arise and interfere with the global modes.

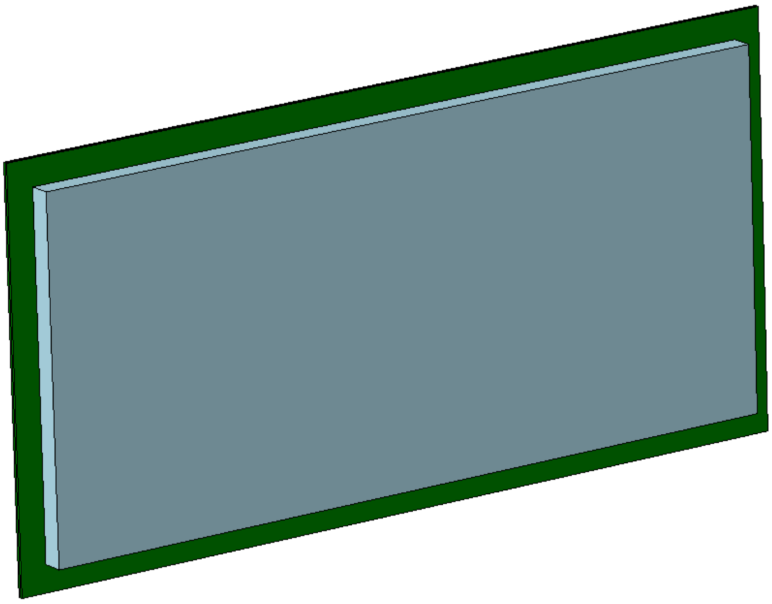


Figure 8.2: Wall panel enclosed by connector zones

It is the in-plane stiffness that it is the most important contribution from the walls to the global stiffness of the structure. A possible solution to the problem with local bending modes in the walls could then be to keep the thickness of the section high, and reduce the material parameters of the material. Reducing the material stiffness instead of the thickness would cause the same reduction of the in-plane stiffness, while out-of-plane bending stiffness will be reduced less than if the thickness were to be reduced. This is because the in-plane stiffness is proportional with the thickness, t, while the bending stiffness is proportional with the thickness cubed, t^3. It could also be argued that altering the shear moduli would be a better way of representing the connections than to change the thickness Young's moduli of the material. Solving the issue with spurious local modes is of great importance if the model should be used for model updating with higher modes as target variables, since the spurious modes interfere with the real modes.

8.2.3 Using Excel for Parameter Input

Relatively early in the work with the parametric model it became clear that the number of input variables to the model was going to be substantial and that writing the input directly into the Python script was not going to be a good solution. By moving all the input variables to a separate excel file the process of setting up a model became much more tidy. Also, more people have experience with Excel than with programming and editing code files, thus using a well known interface might make it easier for inexperienced programmers to use the model.

Also considering that Excel is easily integrated in Isight routines for parameter studies, model updating, optimization etc., the choice of gathering all the input files in a single Excel file must be considered successful. The only real downside is that installing the Python package needed to read Excel files might be cumbersome, however a thorough step-by-step installation guide is provided as a part of Appendix A.

8.2.4 Isight

To produce the results presented in this thesis over 600 analyses have been run in Abaqus, plus at least as many who did not make it into the thesis. It is obvious that running over a thousand analyses manually, over the course of a few weeks after the scripts were ready, would be impossible. Isight has shown to be an extremely useful tool for running many analyses automatically. The native Abaqus component in Isight is not possible to use with the way the model is programmed, however running the scripts using the Simcode component instead have proved to work flawlessly, albeit the setup is more complicated. Isight integrates well with Excel, which makes setting up the updating of the input variables between each of the iterations relatively simple.

An issue with Isight is that the software is not nearly as well documented, as for instance Abaqus is. The lack of documentation meant that the learning process to a large extent was based on trial and error, and therefore required a significant amount of time. To make this process a little easier for anyone that may want to try running a similar analysis to one featured in this thesis, a step-by-step guide is provided in Appendix A.

8.2.5 Damping Estimates and Wind Loads

The Eurocode wind load calculation and the corresponding estimation of acceleration have been tested thoroughly in chapter 7. The analyses performed in this thesis have been successful, the damping estimation method implemented worked well with and gave relatively accurate results when the accuracy where tested in subsection 7.3.1, the same has to be said for the frequency estimation.

However, as Mjøstårnet is a highly symmetric structure, especially for the wind direction tested, this causes the vibration mode of interest to be almost a pure translational/bending mode without any torsion. The way the free vibration is initialized by a uniform impulse load along the upper edge of the structure, combined with the sampling of displacement at the centre of the top level, is a good approach for exciting and capturing pure translational modes.

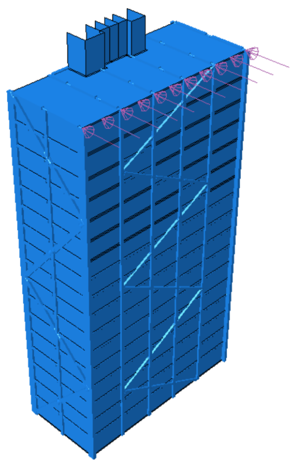


Figure 8.3: Impulse load along upper edge of structure

If the mode includes e.g. torsion, the chosen approach may not be able to capture the mode in a satisfactory manner and the time history would likely have more than one local peak per cycle of vibration, as seen in the plot in Figure 8.4. Since both the frequency and damping estimates relies on the peaks of the time history (ref. section 7.1), and the peak finding algorithm implemented are very simple, the resulting estimates would be completely wrong.

One way to make the script able to deal with time histories who is not a perfect

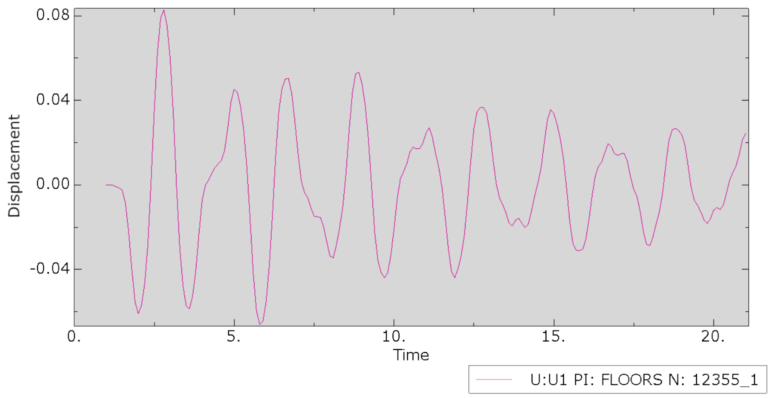


Figure 8.4: Time history with more than one peak per cycle

sinusoidal (although not as messy as shown in Figure 8.4) is to implement a more sophisticated peak finding algorithm, with the ability to filter out small local peaks based on e.g. prominence. In Abaqus 2020 the Scipy package should be supported [53], this package includes an advanced peak finding algorithm with filtering capabilities, called *scipy.signal.find_peaks()*. However to be able to deal with modes dominated by torsion or higher order bending, both the excitation and sampling of deformations in the free vibration step have to be changed. Alternatively an entirely new method of estimating damping and frequency can be implemented, for instance the frequency can relatively easy be taken directly from the frequency step already implemented in the analysis.

8.2.6 Mode Shape Comparison

The only criterion used for the model updating performed in this thesis are the frequencies of the different modes, before the direction and shape of the modes are checked visually after the updating is completed. However after more thorough experiments are performed at Mjøstårnet more detailed information about the mode shapes will be available. If the mode shapes is taken into the model updating as a criterion along with the frequencies, one could be much more certain that the identified parameters resemble the properties of the actual structure.

A possible method for comparing mode shapes, that is also relatively a straight forward method to implement, is using the modal assurance criterion (MAC). The modal assurance criterion is a measure of the similarity between a pair of modes, the result is a scalar (matrix if more than one pair of modes) with values between 0 (no similarity) and 1.0 (full similarity). The definition of the MAC between a set of analytical mode shape vectors $\boldsymbol{\phi}_A$ and set of measured mode shape vectors

$\boldsymbol{\phi}_X$ are given in Equation 8.1 [54].

$$MAC(r,q) = \frac{|\ \boldsymbol{\phi}_{A,r}^T \boldsymbol{\phi}_{X,q}\ |^2}{(\boldsymbol{\phi}_{A,r}^T \boldsymbol{\phi}_{A,r})(\boldsymbol{\phi}_{X,q}^T \boldsymbol{\phi}_{X,q})} \tag{8.1}$$

where:

$\boldsymbol{\phi}_{A,r}$ = Analytical modal vector, mode r
$\boldsymbol{\phi}_{X,q}$ = Experimental mode vector, mode q

Note since the analytical mode shape vector usually contains data for many more DOFs than the measured one, it is necessary to either reduce the analytical or add interpolated values to the measured vector.

8.2.7 Making the Model More General

The parametric model have been developed with Mjøstårnet and other similar buildings in mind. However, during the entire process the design philosophy have been to make the model as general as possible, to allow for future use of the model on other types of buildings as well. Due to the limited time available and to make the model as clear as possible the main scripts, who makes use of the functions written in the other scripts, and input file is somewhat limited to modelling a specific type of building. However we strongly believe that most of the functions written are compatible for use in other types of buildings, with little or no modification. Below follows a list of possible changes to make the model more general:

- The script in its current state is unsuitable for generating non-rectangular buildings. It might work for some configurations, but the function would be unstable and errors must be expected.
- To limit the amount of input parameters to the script there are a few predefined groups of cross sections for the beams, columns and diagonals. If one should wish to change or add more groups of cross sections, it can be done by changing the set definitions in the script and adding the new groups of sets to the input file.
- The model currently assumes that there is floors on every level of the building. Changing this assumption should be doable, for a user with coding experience, directly in the python-script, but the option is not accessible trough the Excel input file.

- There is no option to not generate the exterior walls of the structure. The alternative is to generate the model automatically by use of the script, before removing the selected walls manually in the GUI of Abaqus CAE.
- Currently the stiffness and damping properties are assumed to be identical for the entire foundation. Since the sensitivity study proved that the foundation stiffness are one of the most influential parameters for the dynamical behavior of the structure, it could be interesting to see the effects of being able to assign different stiffness to different parts of the foundations. Adding this option should be straight forward, and is achieved by modifying a the input file and a couple of functions in the script.
- At the current state, the model requires diagonals to be included. In fact, both of the diagonals that can be defined are required. As the model has the possibility of introducing horizontal stiffness in other ways as well, this requirement can be removed in order to allow for more types of buildings to be modelled. In addition, the model only allows for two diagonals to be defined, one placed in each of the vertical principle planes. There are examples of timber buildings, such as "Treet" in Bergen, where more diagonals must be included in order for it to be correctly modelled.
- In order to make it possible to study the damping parameters of floor-to-frame connections, a connection-zones representing these connections must be added to the model.

Chapter 9

Conclusion and Recommendations for Further Work

9.1 Conclusion

The parametric model developed in this thesis has proved to be a powerful tool. It made it possible to conduct a study of how various stiffness parameters influence the frequencies of the first three modes of vibrations of Mjøstårnet. The corresponding mode shapes of mode 1 and 2 were bending in the transverse and longitudinal directions, while mode 3 was torsional. The following parameters were found to have the greatest impact:

- Vertical stiffness of foundations
- Material stiffness of timber frame
- Axial stiffness of connections in the diagonal bracing system
- Rotational stiffness of connections in the timber frame
- Stiffness of exterior walls

A surprising observation was the low influence on the frequencies from the stiffness of the floors of the building.

Tulebekova *et al.* [33] showed that the numerical model Sweco produced during design of Mjøstårnet underestimated the fundamental frequencies and that the two first mode shapes was predicted to be in the opposite direction of what was measured. The results of the sensitivity study in this thesis showed that variations in both the vertical foundation stiffness and the stiffness of the exterior walls would cause the two first mode shapes to switch directions. A possible reason for the wrong estimations made by Sweco can therefore be a combination of exterior walls not being included in the model and an underestimation of the foundation stiffness.

A model updating procedure was used in order to see if it was possible recreate the first three fundamental frequencies of the building accurately and thus find good estimations for the parameters. While it was possible to find parameters that made the model match all the measured frequencies accurately, the procedure was repeated three times, all of which resulted in different parameters. This show that three frequencies alone is not enough in order to use the model to make good predictions of the parameter's of the building. However, if more frequencies can be measured and included together with the mode shapes as targets for the model updating, it is likely that reliable and accurate estimations of the real parameters can be made using the model.

The parametric model also includes possibility for running wind-load analysis based on the Eurocode. A parameter study proved that the acceleration response of the structure is highly dependant on the damping and mode shape of the structure, as well as the wind velocity. A quick comparison with the accelerations measured at different wind velocities by Tulebekova et al. [33] was made and showed relatively good correlation between the calculated and measured results.

9.2 Recommendations for Further Work

An extensive test of "Mjøstårnet" involving a "shaker" and detailed instrumentation is planned as a part of the DynaTTB project [2]. The test will provide measured frequencies for many more modes, as well as information about mode shapes, damping properties etc. that can be added to the list of targets for the model updating and improve the accuracy and certainty of the results considerably. With more output/target variables the number of input variables may also be increased.

During the analyses conducted in the study some weaknesses was discovered in the parametric model. One of which was the spurious local modes that would occur in shell members when the thickness of the connection-zones was set too low. Because of this, very low stiffness values in the connection-zones could not be studied. Another problem caused by this is that the model in its current state, only is able to produce the first three fundamental frequencies reliably, as the spurious modes may interfere with the higher modes. Solving the issue with spurious local modes is of great importance if the model should be used for model updating with higher modes as target variables. Possible solutions to this is discussed in subsection 8.2.2.

In order to be able to study other timber buildings, such as Treet in Bergen, it is likely that a more general model is required. Some suggestions for changes that can be implemented to the model to make it more general is listed in subsection 8.2.7. However, it is strongly believed that the code developed in this thesis can be used as a foundation for further development.

To get more accurate estimations of the response of the structure to wind actions, the already implemented option of specifying a time history for the wind load should be further developed and put in to use, as an alternative to the Eurocode-estimations.

Bibliography

[1] Klima- og miljødepartementet. (13th Mar. 2020). Klimaendringer og norsk klimapolitikk, Regjeringen.no, [Online]. Available: https://www.regjeringen.no/no/tema/klima-og-miljo/innsiktsartikler-klima-miljo/klimaendringer-og-norsk-klimapolitikk/id2636812/ (visited on 05/11/2020).

[2] DynaTTB consortium. (2019). Dynamic response of tall timber buildings under service load, [Online]. Available: https://www.dynattb.com/ (visited on 11/05/2020).

[3] Statsbygg, 'Tre for bygg og bygg i tre: Kunnskapsgrunnlag for økt bruk av tre i offentlige bygg', Analysedokument fra Strategi- og utviklingsavdelingen, 1st Mar. 2013. [Online]. Available: https://www.regjeringen.no/no/dokumenter/tre-for-bygg-og-bygg-i-tre/id721773/.

[4] G. Glasø, 'Tre og brann', Treteknisk, Oslo, FOKUS på tre 37, Feb. 2012. [Online]. Available: http://trefokus.no/resources/filer/fokus-pa-tre/37-Tre-og-brann.pdf.

[5] P. Bernhard and P. F. Jørgensen, 'Byggsektorens klimagassutslipp', KanEnergi AS, Oslo, 19th Apr. 2007. [Online]. Available: http://www.byggemiljo.no/wp-content/uploads/2015/01/Notat-klimagassutslipp-fra-byggsektoren21des06rev190407.pdf.

[6] Ø. Selvig, 'Økt bruk av tre i offentlige bygg - klimagassvirkninger', Civitas, Oslo, Notat, 31st Jan. 2013. [Online]. Available: https://www.statsbygg.no/globalassets/files/publikasjoner/rapporter/oktbruktreoffbygg-civitas2013.pdf.

[7] Treindustrien, TreFokus, Skogeierforbundet and Treteknisk, 'Treindustriens lille grønne', Oslo, May 2013. [Online]. Available: http://www.trefokus.no/resources/Treindustriens-lille-gronne.pdf.

[8] Swedish Wood, *Design of timber structures, Volume 1*, 2nd ed. Stockholm: Swedish Forest Industries Federatioin, 2016, ISBN: 978-91-980304-8-8.

[9] K. A. Malo, 'Anisotropy in wooden materials', Lecture Notes (TKT 4212 - Timber Structuers 2), Department of Structal Engineering, NTNU, Trondheim, Norway, 28th Aug. 2018.

[10] 'Eurokode 3: Prosjektering av stålkonstruksjoner. del 1-1: Allmenne regler og regler for bygninger', Standard Norge, Lysaker, NS-EN 1993-1-1:2005 +A1:2014+NA:2015, 2015.

[11] 'Eurokode 2: Prosjektering av betongkonstruksjoner. del 1-1: Allmenne regler og regler for bygninger', Standard Norge, Lysaker, NS-EN 1992-1-1:2004 +NA:2008, 2008.

[12] 'Konstruksjonstrevirke - fasthetsklasser', Standard Norge, Lysaker, NS-EN 338:2016, 2016.

[13] N. Labonnote, *Damping in Timber Structures*. Department of Structal Engineering, NTNU, Trondheim, Norway: Phd. Thesis, 2012, ISBN: 978-82-471-3836-6.

[14] Jayamon Jeena R., Line Philip and Charney Finley A., 'State-of-the-art review on damping in wood-frame shear wall structures', *Journal of Structural Engineering*, vol. 144, no. 12, 2018. DOI: 10.1061/(ASCE)ST.1943-541X.0002212.

[15] K. A. Malo, R. Abrahamsen and M. Bjertnæs, 'Some structural design issues of the 14-storey timber framed building "treet" in norway', *European Journal of Wood and Wood Products*, vol. 74, pp. 407–424, 2016. DOI: 10.1007/s00107-016-1022-5.

[16] M. Williams, *Structural Dynamics*. Boca Raton, USA: Taylor & Francis, 2016, 256 pp., ISBN: 978-0-415-42732-6.

[17] A. Harish. (16th Jul. 2018). Why the tacoma narrows bridge collapsed: An engineering analysis, SimScale Blog, [Online]. Available: https://www.simscale.com/blog/2018/07/tacoma-narrows-bridge-collapse/ (visited on 30/01/2020).

[18] A. K. Chopra, *Dynamics of Structures*, 4th ed. Boston: Prentice Hall, 2012, 979 pp., ISBN: 978-0-13-307269-3.

[19] M. Liu and D. G. Gorman, 'Formulation of rayleigh damping and its extensions', *Computers & Structures*, vol. 57, no. 2, pp. 277–285, Oct. 1995. DOI: 10.1016/0045-7949(94)00611-6.

[20] C. T. Yeh, B. J. Hartz and C. B. Brown, 'Damping sources in wood structures', *Journal of Sound and Vibration*, vol. 19, no. 4, pp. 411–419, Dec. 1971. DOI: 10.1016/0022-460X(71)90612-2.

[21] Y. Tamura and A. Kareem, Eds., *Advanced Structural Wind Engineering*, Tokyo: Springer Japan, 2013, ISBN: 978-4-431-54337-4. DOI: 10.1007/978-4-431-54337-4.

[22] E. N. Strømmen, *Theory of bridge aerodynamics*, 2nd ed. Berlin: Springer, 2010, 302 pp., ISBN: 978-3-642-13659-7.

[23] 'Eurokode 1: Laster på konstruksjoner: Del 1-4 : Allmenne laster. vindlaster', Standard Norge, Lysaker, NS-EN 1991-1-4:2005+NA:2009, 2009.

[24] A. Feldmann, H. Huang, W.-S. Chang, R. Harris, P. Dietsch, M. Gräfe and C. Hein, 'Dynamic properties of tall timber structures under wind-induced vibration', presented at the WCTE World Conference on Timber Engineering 2016, Vienna, Austria, Aug. 2016. [Online]. Available: `https://www.semanticscholar.org/paper/Dynamic-properties-of-tall-timber-structures-under-Feldmann-Huang/8f54865d1d737c8da6653ca2af1ce81c2258986f`.

[25] 'Bases for design of structures - serviceability of buildings and walkways against vibrations', Geneva, Switzerland, ISO 10137:2007, 2007.

[26] A. Talja and L. Fülüp, 'Evaluation of wind-induced vibrations of modular buildings', VTT Technical Research Centere of Finland Ltd, Helsinki, Customer Report VTT-CR-03593-16, 2016. [Online]. Available: `https://www.vttresearch.com/sites/default/files/julkaisut/muut/2016/VTT-CR-03593-16.pdf` (visited on 01/04/2020).

[27] K. Bell, *An Engineering Approach to Finite Element Analysis of Linear Structural Mechanics Problems*. Bergen: Fagbokforlaget, 2014, ISBN: 978-82-321-0268-6.

[28] *Abaqus 6.11 - Analysis User's Manual*. Providence, USA: Dassault Systèmes, 2011.

[29] *Abaqus 6.11 - Theory Manual*. Providence, USA: Dassault Systèmes, 2011.

[30] R. Abrahamsen, 'Mjøstårnet - 18 storey timber building completed', presented at the Internationales Holzbau-Forum, Garmisch-Partenkirchen, Germany, 2018. [Online]. Available: `https://www.moelven.com/globalassets/moelven-limtre/mjostarnet/mjostarnet---18-storey-timber-building-completed.pdf` (visited on 05/05/2020).

[31] R. Abrahamsen, 'Mjøstårnet - construction of an 81 m tall timber building', presented at the Internationales Holzbau-Forum, Garmisch-Partenkirchen, Germany, 2017. [Online]. Available: `https://www.moelven.com/globalassets/moelven-limtre/mjostarnet/mjostarnet---construction-of-an-81-m-tall-timber-building.pdf` (visited on 05/05/2020).

[32] 'Trekonstruksjoner - limtre og limt laminert heltre - krav.', Standard Norge, Lysaker, NS-EN 14080:2013+NA:2016, 2016.

[33] S. Tulebekova, K. A. Malo, A. Rønnquist, P. Nåvik and M. Bjærtnes, 'Identification of structural damping from ambient vibrations in an 18-storey timber building in norway [paper to be submitted]', Department of Structal Engineering, NTNU, Trondheim, Norway, 2020.

[34] S. Tulebekova, 'Mjøstårnet dynamics', Department of Structal Engineering, NTNU, Trondheim, Norway, Internal Presentation, 20th Feb. 2020.

[35] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[36] Dassault Systèmes, *Abaqus 2016 - Scripting Reference Guide*. Providence, USA: Dassault Systèmes, 2015.

[37] Dassault Systèmes, *Isight - automate design exploration and optimaztion*, 2014. [Online]. Available: https://www.3ds.com/fileadmin/PRODUCTS-SERVICES/SIMULIA/RESOURCES/simulia-isight-brochure.pdf (visited on 15/05/2020).

[38] K. A. Malo, 'Guidance meetings', Department of Structal Engineering, NTNU, Trondheim, Norway, 2020.

[39] I. Utne, 'Numerical models for dynamic properties of a 14 storey timber building', Master Thesis, NTNU, Trondheim, 2012. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/237175 (visited on 17/01/2020).

[40] H. Liven and Moelven Limtre AS, 'Treet i bergen og mjøstårnet i brumunddal; høyhus i tre - utfordringer', Bygg Reis Deg, Presentation, 2017.

[41] Metsä Wood, 'Kerto manual - mechanical properties', Metsä, Finland, Feb. 2017. [Online]. Available: https://www.metsawood.com/global/Tools/kerto-manual/Pages/Kerto-manual.aspx.

[42] S. Nesheim, 'Python script for abaqus for closed hollow sections (PSACHS)', Department of Structal Engineering, NTNU, Trondheim, Norway, Nov. 2019.

[43] H. Unterwieser and G. Schickhofer, 'Characteristic values and test configurations of CLT with focus on selected properties', presented at the Focus Solid Timber Solutions - European Conference on Cross Laminated Timber (CLT), Graz, Austria: University of Bath, 2013, pp. 53–73.

[44] 'Egenlaster for bygningsmaterialer, byggevarer og bygningsdeler', Sintef, Trondheim, Byggforskserien 471.031, 2013.

[45] 'Eurocode 1: Actions on structures. part 1-1: General actions. densities, self-weight, imposed loads for buildings', Standard Norge, Lysaker, NS-EN 1991-1-1:2002+NA:2008, 2008.

[46] 'Eurocode 0: Basis of structural design', Standard Norge, Lysaker, NS-EN 1990:2002+A1:2005+NA:2016, 2016.

[47] A. Rønnquist, 'Lecture 9 - damping of structures', Lecture Notes (TKT 4201 - Structural Dynamics), Department of Structal Engineering, NTNU, Trondheim, Norway, 2019.

[48] 'Guidelines for the evaluation of the response of occupants of fixed structures, especially buildings and off-shore structures, to low-frequency horizontal motion (0,063 to 1 hz)', Geneva, Switzerland, ISO 6897:1984, 1984.

[49] 'Eurokode 5: Prosjektering av trekonstruksjoner. del 1-1: Allmenne regler og regler for bygninger.', Standard Norge, Lysaker, NS-EN 1995-1-1:2004 +A1:2008+NA:2010, 2010.

[50] J. M. Branco, P. J. S. Cruz and M. Piazza, 'Experimental analysis of laterally loaded nailed timber-to-concrete connections', *Construction and Building Materials*, vol. 23, no. 1, pp. 400–410, Jan. 2009. DOI: 10.1016/j.conbuildmat.2007.11.011.

[51] F. Solarino, L. Giresini, W.-S. Chang and H. Huang, 'Experimental tests on a dowel-type timber connection and validation of numerical models', *Buildings*, vol. 7, p. 116, Dec. 2017. DOI: `10.3390/buildings7040116`.

[52] R. Tomasi, A. Crosatti and M. Piazza, 'Theoretical and experimental analysis of timber-to-timber joints connected with inclined screws', *Construction and Building Materials*, vol. 24, no. 9, pp. 1560–1571, Sep. 2010. DOI: `10.1016/j.conbuildmat.2010.03.007`. (visited on 06/06/2020).

[53] C. Obbink-Huizer. (9th Dec. 2020). Abaqus 2020: What's new?, [Online]. Available: `https://info.simuleon.com/blog/abaqus-2020-whats-new` (visited on 08/06/2020).

[54] M. Pastor, M. Binda and T. Harčarik, 'Modal assurance criterion', *Procedia Engineering*, Modelling of Mechanical and Mechatronics Systems, vol. 48, pp. 543–548, Jan. 2012. DOI: `10.1016/j.proeng.2012.09.551`.

Appendix A

Parametric Model - User Guide

This document gives a detailed guide on how to create a parametric model using the scripts developed in the master thesis "A Parametric Study of Tall Timber Buildings" by Lars Håkon Wiig and Daniel Hjohlman Reed. The document also covers how to run analyses using the model. Assumptions and limitations of the model are covered in the thesis.

A.1 Prerequisites

Before running the model a few prerequisites must be fulfilled:

- Microsoft Excel must be installed
- Simulia Abaqus must be installed, preferably version 2019
- Simulia Isight (optional)
- OpenPyXL must be downloaded and added to Abaqus (subsection A.1.1)
- File paths must be updated inside the script (subsection A.1.2)

A.1.1 Installing OpenPyXl

The script relies on a Python package called *OpenPyXl* to be able to read the data from the input file created in Excel. OpenPyXl is a package that is not included in a standard installation of Python. Normally downloading and installing packages in Python is relatively straight forward, however this is not the case with the Python installation featured in Abaqus. So instead of a straight forward installation, getting OpenPyXl to work demands some extra steps, described below:

- **Method 1 (The easy way):**
  This method is tested for Abaqus 2017 and 2019, and will likely work for other Abaqus versions as well. This method is identical to "method 2" expect for that the files that needs to be downloaded and installed in "method 2" are already provided in the digital appendix.

  1. Unzip the .zip archive called "OpenPyXl_files.zip" featured in the digital appendix, and copy the content (not the folder itself) of the folder.
  2. Locate the "site-packages" folder containing the packages featured in the Abaqus Python installation, the path should be something like: `C:/SIMULIA/CAE/2019/win_b64/tools/SMApy/python2.7/Lib/site-packages`
  3. Paste the content copied in step 1 into the folder located in step 2.
  4. Check that OpenPyXl is installed by opening Abaqus and typing the following command into the python interpreter inside Abaqus:

     ```
     >>> import openpyxl
     ```

     Press enter and if no error messages are returned the installation should be successful.

- **Method 2 (The complicated way):**
  If an Abaqus version running a Python version not compatible with OpenPyXl 2.6.4 is used, or for any other reason method 1 does not work, a different version of OpenPyXl can be downloaded from the original source rather than the digital appendix.

  1. Begin by checking the Python version included in the Abaqus version by typing the following into the python interpreter in Abaqus, note that at the time of writing all Abaqus versions uses old 2.7.x versions of Python instead of the newer 3.x.x:

     ```
     >>> import sys
     >>> print(sys.version)
     ```

  2. Download and install the same version of Python as a standalone installation on your computer. The necessary installation files are found at `https://www.python.org/downloads/`.
  3. Download and add OpenPyXl to the Python installation installed in the previous step. See `https://openpyxl.readthedocs.io/en/stable/index.html#installation` and `https://packaging.python.org/tutorials/installing-packages/` for installation instructions. Make sure that you install a version of OpenPyXl that is compatible with the installed version of Python, as the newer versions of OpenPyXl does not work with Python 2.x.x. If using pip to install the package the version (of OpenPyXl) can be specified by using the following command:

     ```
     pip install openpyxl==x.x.x
     ```

where x.x.x is the desired version number of OpenPyXl. At the time of writing this instructions the final version of OpenPyXl that can be used with Python 2.7.x is 2.6.4. Note that the files needed for installing version 2.6.4 are provided in the digital appendix as described in "method 1".

4. Open the (standalone, not inside Abaqus) python interpreter and type the following command:

```
>>> import openpyxl
```

   Press enter and if no error messages are returned the installation should be successful.

5. After OpenPyXl is installed to the standalone Python version locate the "site-packages" folder containing the installed libraries (for the standalone version), the path to this folder is usually something like: `C:\Python27\Lib\site-packages`.
6. Copy all the files and folders with names related to either "xmlfile", "jd_cal" or "openpyxl", similar to the files in Figure A.1.

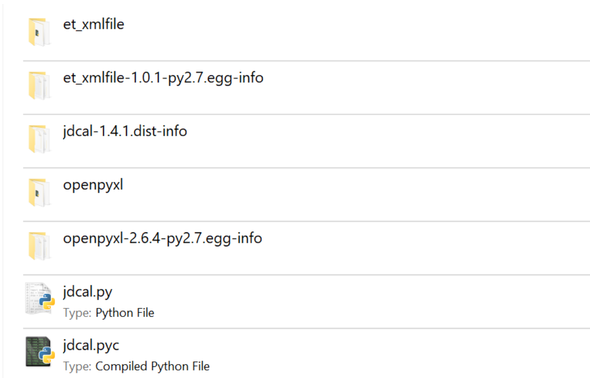


Figure A.1: Folders and files to copy. Note: Might vary depending on the version of OpenPyXl

7. Locate the "site-packages" folder containing the packages featured in the Abaqus Python installation, the path should be something like: `C:/SIMULIA/CAE/2019/win_b64/tools/SMApy/python2.7/Lib/site-packages`
8. Paste the content copied in step 6 into the folder located in step 7.
9. Check that OpenPyXl is installed by opening Abaqus and typing the following into the Python interpreter inside Abaqus:

```
>>> import openpyxl
```

   Press enter and no error messages should be returned.

A.1.2 Preparing the Scripts

Before running the code, each script needs to be updated with the path to the folder containing all the scripts. Open the scripts in a code editor or simply a basic text editor, e.g Microsoft Notepad. Near the top of every script the following lines can be found:

```
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'
```

Replace the path with the real path to the folder where all the scripts are located. Remember to keep the quotation marks enclosing the path.

In addition to updating the path to the folder containing all the scripts, the path to the the input file (Excel file) and the working directory must be specified in the main scripts. In the files *TTB_3D.py* and *TTB_3D_EC_wind.py* locate the following lines:

```
# -------------- Input file/folder paths -------------
# All the locations specified must exist (i.e folders must be
 ↪  created BEFORE running the script)
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'
# Path to the Excel-file containing the input:
inputFile =
 ↪  'C:\\Users\\username\\TTBParametricModel\\TTB_input.xlsx'
# Path to Abaqus working directory (all result files will be
 ↪  stored here):
workDir = 'C:\\temp'
```

Replace the paths in the scripts with the relevant paths, all the folders specified in the paths must be created before running the scripts, Abaqus/Python will not create them automatically. Remember to keep the quotation marks enclosing the path.

A.2 Setting Up the Input File

This section goes through the Excel workbook sheet-by-sheet. The purpose is to explain the input that is used in the model. The Excel file is a part of the digital appendix delivered directly to prof. Malo at the Department of Structural Engineering at NTNU. Note that the input shown in the screenshots featured is for a fictitious building that is not related to Mjøstårnet.

A.2.1 General Remarks

The following list points out a few general remarks for use of input file:

- Only add input to yellow cells. Some red cells turn yellow if depending on the previous user input, and can then be modified.
- Cells that are either, red, white or grey should _not be modified.
- All input should be inserted to the table starting from the top row and/or left column. No rows should be left empty between two rows that contain input.
- Do not add cells, rows or columns to the file. (Unless the appropriate changes have been applied to the script)
- Some of the inputs are in the form of questions. In these cases the user can choose between 1 and 0, which means Yes and No, respectively.
- It is recommended to fill out the sheets in the same order as they appear in the user guide.

A.2.2 Units

Abaqus lets the user specify the input in whatever units they want as long as they are consistent, however we strongly recommend using SI-units as listed in Table A.1.

Table A.1: Recommended units

Length	Force	Mass	Time	Stress	Energy	Density	Angle (Rot. DOFS)
m	N	kg	s	Pa	J	kg/m^3	rad

A.2.3 Coordinate System

The coordinate system used for the model is oriented with the xz-plane as the horizontal plane. The y-axis is pointing in positive vertical direction. The script only allows for internal beams to span in one direction, the x-direction. The x-direction is typically along the short side of a building, and will hereafter be referred to as the transverse direction. The z-direction is along the long side of the building, hereafter referred to as the longitudinal direction. The orientation of the coordinate system is illustrated in Figure A.2.

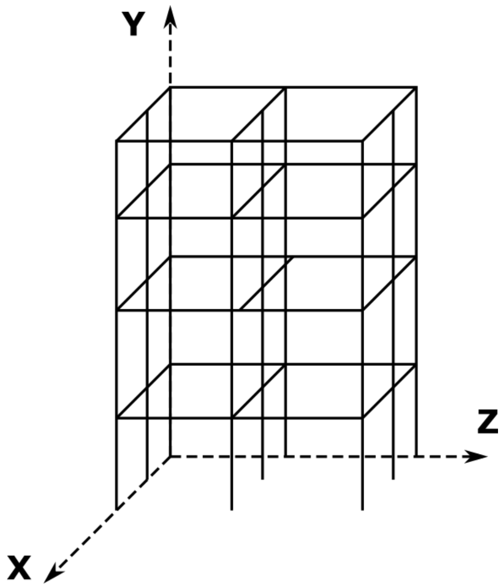


Figure A.2: Orientation of coordinate system

A.2.4 Grid

The geometry of the entire model is based on the grid system. By defining the grid, the geometry of the frame is also automatically defined. The grid is defined in two separate sheets, the "Grid (XZ)" sheet (Figure A.3) and the "Grid (Y)" sheet (Figure A.4). In the "Grid (XZ)" sheet, the positions of the x- and z-grid lines are specified in the position rows. The grid lines are given a reference number, starting from zero. The two position vectors are then creating a matrix where every element is indicating a grid line intersection. A column is placed at every grid line intersection that is marked with 1. If it instead reads 0, there will not be placed a column. The example input in Figure A.3 results in the column placement illustrated in Figure A.5.

The "Grid (Y)" sheet defines the levels of the grid. In this sheet, the vertical coordinate of the levels is the only thing that should be specified. Each level is given an index, starting from 0. At every level, except from level 0, beams are added to the frame. Beams spanning in x-direction will be placed at every grid line, while beams spanning in z-direction will only be placed at the two outermost grid lines.

	Z-Axis	0	1	2	3	4	5	6	7	8
X-Axis	Position	0	10	20	30	40	50			
0	0	1	1	1	1	1	1			
1	5	0	1	1	1	1	0			
2	10	1	0	0	0	0	1			
3	15	0	1	1	1	1	0			
4	20	1	1	1	1	1	1			
5										
6										
7										
8										
9										
10										
11										
12										

Figure A.3: Grid input for the horizontal plane (xz-plane)

Level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Y-Coordinates	0.00	4.00	8.00	12.00	16.00	20.00	24.00	28.00	32.00	36.00	40.00	44.00	48.00						

Figure A.4: Grid input for the vertical direction (y-direction)

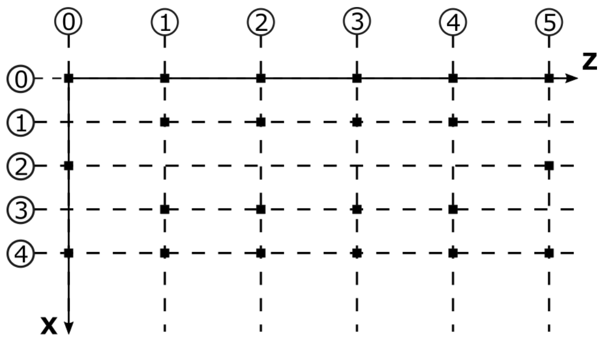


Figure A.5: Column placement based on input in Figure A.3

A.2.5 Diagonals

The geometry of the diagonals are defined in the "Diagonals" sheet (Figure A.6). The script requires the model to include two types of diagonals: one in both the longitudinal (z) and transverse (x) directions. The sheet have two input rows. New rows should not be added. The "LongEdgeDiagonals" and the "ShortEdgeDiagonals" specify the parameters of the longitudinal and transverse diagonals, respectively.

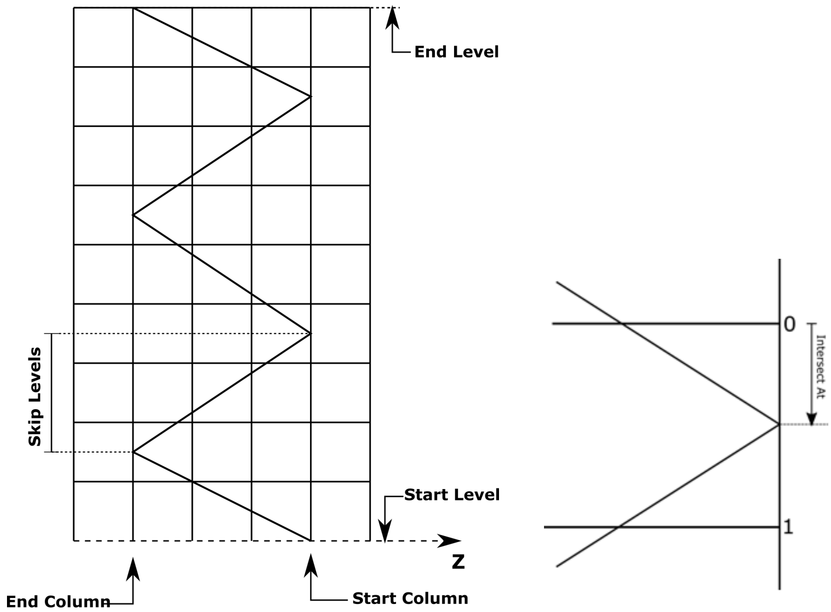
The "Plane" input should not be altered, as it is preset to orient the diagonals correctly. The input of the "Axis" column defines the grid lines, at which the diagonals should be placed. The input should be one or more of the grid lines indices defined in the "Grid (XZ)" sheet. If the diagonal is to be placed at more than one grid line, the grid line indices should be separated by a semi-colon, see Figure A.6. The "Start Level" and "End Level" inputs specify at what levels the diagonal should start and end, respectively. Note that the start level should always have a lower index than the end level. The "Start Column" and "End Column" inputs specify the the columns that the diagonal should be placed between. The input should be grid line indices. Note that the "Start Column" define the direction of the diagonal, and does not necessarily have to be the lowest index (see Figure A.7).

The "Skip Levels" input specify the number of levels the diagonal should span across before it changes direction. If the number of floors that the diagonal span across varies throughout the height of the building a list of values separated by semi-colons, can be used instead of a constant. In this case the first value specify the number of floors that is spanned across between the start of the diagonal and the first turning point, the second value between the first and the second turning point etc. The example in Figure A.7a, has a constant "Skip Levels" input of 2.

The "Intersect At" parameter defines at what height the diagonal should intersect with the start and end columns. The input can be of any value in the interval [0,1]. If the input is set to 0 the turning point will be placed at the level specified by the "Skip Levels" parameter, if the input is set to 1 it will be placed at the level below and if it is 0.5 it will be placed in the middle of the two levels. If the "Start Level" is set to 0, the diagonal will start at this level regardless of "Intersect At" input. Likewise, if "End Level" is set to the upper level of the building, diagonal will end at the top level. The "Intersect At" parameter is illustrated in Figure A.7b. The example in Figure A.7a uses an "Intersect At" value of 0.5.

Name/Description	Plane	Axis	Start Level	End Level	Start Column	End Column	Skip Levels (If non-uniform: separate by semicolon)	Intersect At
LongEdgeDiagonals	YZ	0;4	0	17	1	4	3	0.5
ShortEdgeDiagonals	XY	0;5	0	17	0	2	4;4;5;4	0

Figure A.6: Diagonals input



(a) Parameter description **(b)** Intersect At

Figure A.7: Illustration of input parameters for "Diagonals" sheet.

A.2.6 Materials

The materials to be used later in the model are defined in the "Materials" sheet (Figure A.8). The script allows the input of isotropic, transverse isotropic and orthotropic linear elastic materials.

Begin by choosing the material type from the drop-down menu in the "Type" column. The cells that needs to be filled out for defining the chosen material type will turn yellow. Values must be inserted in the "Name", "Density" and the highlighted "Stiffness Parameters" columns. The input for the damping parameters are optional. Note that any damping defined in the "Materials" sheet does not apply to the connection zones or segments. Also note that using the composite damping option may lead to issues when used in combination with the other damping types, see subsection A.2.18 and the Abaqus documentation for more information.

												Optional			
			Stiffness Parameters									Rayleigh Damping		Other Damping	
Name	Type	Density	E1	E2	E3	Nu12	Nu13	Nu23	G12	G13	G23	Alpha	Beta	Composite	Structural
Concrete	Isotropic	2400	2,66E+10			0,2									
Timber GL30c	Trans. Isotropic	430	1,30E+10	3,00E+08		0,49		0,64	6,50E+08						
	-														
	-														
	-														
	-														
	-														

Figure A.8: Material input

A.2.7 Add to/Remove From Frame

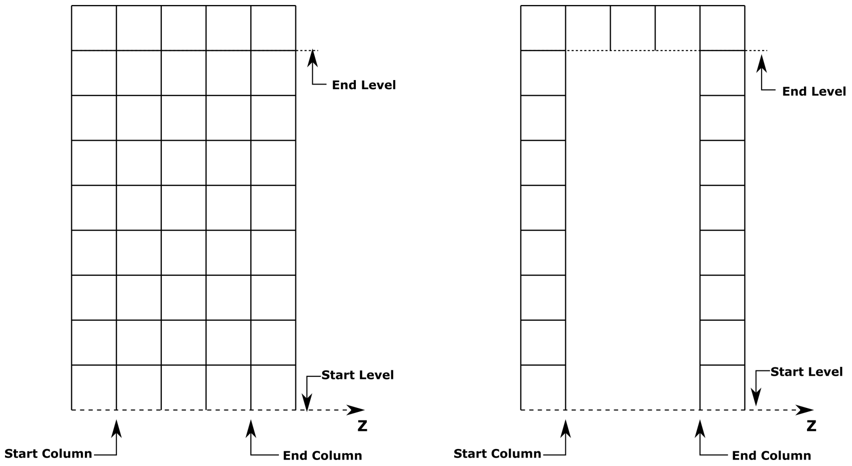
The "Remove From Frame" and "Add To Frame" sheets can be used for altering the original geometry of the timber frame that is defined by the grid. Only beams and columns can be removed and added.

Starting with the "Remove From Frame" sheet (Figure A.9), the general idea of this function is that beams and columns that lays along a specified grid line is removed. The first input is "Parts to be removed". The input is list based, and specify what should be removed. The options are "Beams", "Columns" and "Beams and Columns". The "Plane" input specifies what plane the the parts should be removed from. The two possible inputs are "XY" (transverse plane) and "YZ" (longitudinal plane).

The "Axis" input specifies from what grid lines the parts should be removed. If parts should be removed from multiple grid lines, more than one grid line number could be added, separated by semi-colons. The "Start Level" and "End Level" inputs specify the vertical limits of the area in which the parts will be removed, the input should be level indices. If "Parts to be removed" includes beams, the beams placed at the specified start and end levels will be removed. The "Start Column" and "End Column" define the horizontal limits of the area that the operation is applied to. Note that the start index for both the start level and column must be lower than the end index. Finally, the "Remove start/end columns" should be set to 1 if the start and end columns specified should be removed, and 0 in the opposite case. This input is only relevant if columns are included in the "Parts to be removed" input. The results of the input in Figure A.9, are illustrated in

Parts to be removed	Plane	Axis	Start Level	End Level	Start Column	End Column	Remove start/end columns?
Beams and Columns	YZ	0;3	0	8	1	4	0

Figure A.9: Remove From Frame input



(a) Before **(b)** After

Figure A.10: Example for use of "Remove From Frame" sheet

The "Add To Frame" sheet (Figure A.11), allows for adding individual beams and columns to the frame. The inputs that defines the geometry and the section of the added features are shown in Figure A.11a. Each added feature is defined by one row, and it must be given a unique name in the "Name/Description" column. The "Startpoint" and "Endpoint" inputs define the position of the added feature. The points are specified by coordinates, and can be placed independently of the grid. Note that it is only possible to add features that solely span in either x-, y- or z-direction.

The next input parameters are related to the cross-section. The "Width" and "Height" inputs specify the geometry of the cross-section, and the "Material" input assigns the material. The "Orientation of n1" input defines how the cross-section is oriented in the global coordinate system. The input is a unit vector that specify the n1 direction (see Figure A.12). The vector should be written on the form x;y;z.

	Startpoint			Endpoint			Section							
Name/Description	X	Y	Z	X	Y	Z	Width	Height	Material	Orientation of n1	Area	I_11	I_22	J
Added Beam 1	0	5	8	0	5	12	0.6	0.3	Timber GL30c	1;0;0	1.800E-01	1.350E-03	5.400E-03	3.708E-03
Added Beam 2	0	5	12	0	5	16	0.6	0.3	Timber GL30c	1;0;0	1.800E-01	1.350E-03	5.400E-03	3.708E-03
Added Beam 3	0	5	16	0	5	20	0.6	0.3	Timber GL30c	1;0;0	1.800E-01	1.350E-03	5.400E-03	3.708E-03
Added Column 1	4	0	4	4	60	4	0.8	0.8	Timber GL30c	1;0;0	6.400E-01	3.413E-02	3.413E-02	5.769E-02

(a) Geometry and section input

Connector segment													
Stiffness/Geometry										Damping (Optional)		
		Fraction Of Original Section				Resulting				Rayleigh		Other
Connector segment?	Segment Length	Area	I_11	I_22	J	Area	I_11	I_22	J	Alpha	Beta	Composite
1	0.6	0.2	0.2	0.2	0.2	3.600E-02	2.700E-04	1.080E-03	7.416E-04			
1	0.6	0.2	0.2	0.2	0.2	3.600E-02	2.700E-04	1.080E-03	7.416E-04			
1	0.6	0.2	0.2	0.2	0.2	3.600E-02	2.700E-04	1.080E-03	7.416E-04			
0												

(b) Connector segment input

Figure A.11: Add to frame input

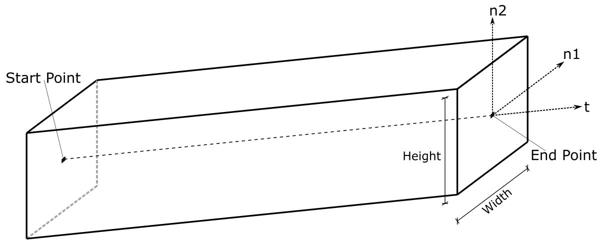


Figure A.12: Orientation of cross-section

The final inputs that need to be defined for every added feature, are related to the connections, or more specifically the connection segments used to model the connections (Figure A.11b). If the "Connector segments?" input is set to 1 connector segments are added to both ends of the added feature. If it is set to 0 it will not be created a connector segment, and the added feature will be rigidly connected to other parts of the frame. The following input parameters are only required for added features where "Connector segments?" is set to 1. These are specified in the same way as in the "Beam Connections" sheet, which is explained in subsection A.2.10.

A.2.8 Shafts

The "Shafts" sheet (Figure A.13) is where the geometry of the shafts of the model is defined. It is required to include one or more shafts in the model in order for the script to run correctly. Start by giving each shaft a unique name in the "Name/Description" column. The "Connect to Building" input is used to specify if the shaft walls should be attached to other parts of the building, thus adding lateral support. The shaft will be connected to the building if this input is set to 1. If it is set to 0, the shaft itself will not be created, but shaft openings will be created in the floors. The position of the shafts in the horizontal (xz-) plane, is defined by coordinates, rather than referring to the grid lines. This is done in the "Start Coordinate" and "End Coordinate" columns.

The "Start Level" and "End Level" parameters specify the top and bottom of the shaft. These inputs are limited to the level indices. However, in the "End Level Offset" column, an offset of the the end level can be specified, allowing the shaft to span to any desired coordinate. The input should be in the chosen length unit for the model. If no offset is desired, the column should be kept blank. Finally, the "Remove Wall" input is also optional. This allows for removing one of the walls from the shaft. The input should be a number between 1-4, which specify which wall should be removed. The numbering of the walls is illustrated in Figure A.14. The column should be left blank if no wall is to be removed.

		Start Coordinate		End Coordinate					
Name/Description	Connect to Building	X	Z	X	Z	Start Level	End Level	End Level Offset	Remove Wall (Optional)
Shaft 1	1	0	10	5	15	0	17	6	4

Figure A.13: Shaft input

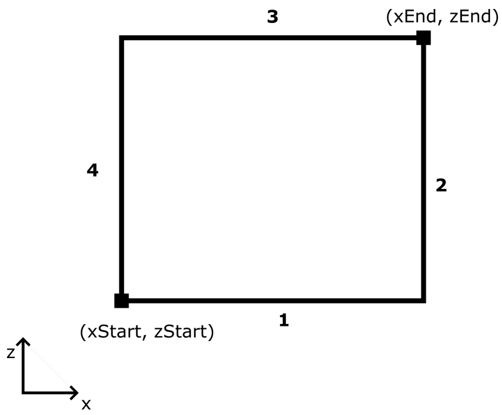


Figure A.14: Shaft wall numbering

A.2.9 Column/Beam/Diagonal Cross Sections

The input sheets for setting the cross sections of the columns, beams, and diagonals are identical. Figure A.15 shows the input sheet for the beams. The "Name" column contains the names of the <u>predefined</u> groups of columns/beams/diagonals (see section 3.3), the names should <u>not</u> be changed. The two following columns are for defining the section width and height respectively. For all beams and diagonals, the cross-section is oriented such that the width is parallel to the horizontal plane. Note that the orientation varies amongst the column groups. Corner and Short Edge columns are oriented with the width along the Z-direction, while Long Edge and Inner columns are oriented with the width along X-direction. Finally choose the material from the drop-down menu in the "Material" column. The material must be defined in the "Materials" sheet before in appears in the drop-down menu. The final remaining columns contains cross sectional properties that are calculated automatically, and should not be changed manually.

Name	a (width)	b (height)	Material	Area	I_11	I_22	J
LongEdgeBeams	0,150	0,250	Timber GL30c	3,75E-02	1,95E-04	7,03E-05	1,76E-04
ShortEdgeBeams	0,200	0,400	Timber GL30c	8,00E-02	1,07E-03	2,67E-04	7,32E-04
InnerBeams	0,200	0,400	Timber GL30c	8,00E-02	1,07E-03	2,67E-04	7,32E-04

Figure A.15: Beam cross sections input

A.2.10 Beam Connections

The principle behind the connections between the diagonals, beams and columns are explained in detail in subsection 3.7.1 of the thesis. Figure A.16 shows the input sheet where the properties of the connector segments are set. Like for the previous sheet the names in the first column are predefined in the code and should not be changed. The length of the connector segments are set in the "Segment Length" column. The next four columns are used for setting the fraction of the original area, 2nd moment of area (about both axes) and the torsional constant of the connector segment. The grey columns shows the resulting properties of the connectors.

		Stiffness/Geometry								Damping (Optional)		
		Fraction of...				Resulting				Rayleigh		Other
Name (Original)	Segment Length	Area	I_11	I_22	J	Area	I_11	I_22	J	Alpha	Beta	Composite
LongEdgeBeams	0,500	0,2	0,2	0,2	0,2	7,50E-03	3,91E-05	1,41E-05	3,52E-05			
ShortEdgeBeams	0,500	0,2	0,2	0,2	0,2	1,60E-02	2,13E-04	5,33E-05	1,46E-04			
InnerBeams	0,500	0,2	0,2	0,2	0,2	1,60E-02	2,13E-04	5,33E-05	1,46E-04			
LongEdgeDiagonals	1,000	0,2	0,2	0,2	0,2	6,13E-02	1,23E-03	1,99E-03	2,56E-03			
ShortEdgeDiagonals	1,000	0,2	0,2	0,2	0,2	1,24E-01	1,01E-02	4,03E-03	9,79E-03			

Figure A.16: Beam connections sheet

Finally the damping parameters of the connections are set. The generalized cross sections used to create the connections in Abaqus does not support the structural damping option, only Rayleigh and composite damping. Note that any damping defined in the "Materials" sheet does not apply to the connection segments. Also note that using the composite damping option may lead to issues when used in combination with the other damping types, see subsection A.2.18 and the Abaqus documentation for more information.

A.2.11 Wall Sections

The cross sections of the walls are set in the "Wall Sections" sheet (Figure A.17). The wall types are predefined inside the code, and the only settings on this sheet are the section thickness and the material. The material must be defined in the "Materials" sheet to show up in the drop-down menu.

Name	Thickness	Material
Walls	0,45	Wall Material
Shaft Walls	0,15	Concrete

Figure A.17: Wall sections input

A.2.12 Floor Sections

The parameters of the floors in the building are defined in the "Floor Sections" sheet (Figure A.18). Start by defining the name, start and end level of the floor type. As many different types of floors as necessary can be created, but each story can only be assigned one type of floor section. A floor must be defined for every level in the building for the model to function properly.

Name	Start Level	End Level	Thickness	Material	Include Outer Beams	Include Connector Segments	Distance between connector segments	Main (E1) Direction
Timber Floors	1	10	0,400	Timber Floor Material	1	1	2,4	Z
Concrete Floors	11	17	0,300	Concrete	0	0		-
Concrete Slab	0	0	0,300	Concrete	0	0		-

Figure A.18: Floor sections input

The script does only support creating homogeneous shell sections. Enter the section thickness and pick a material from the drop-down menu in the "Material" column. The next column gives the option of including (set the value to 1) beams at the outer edges of the floor or not (set the value to 0). The "Include Connector Segments" option can be turned on to create longitudinal connector zones with the spacing specified in the "Distance between connector segments" column, note that the connector zones can only be created in the global z-direction of the model. The final column "Main (E1) Direction" is used to define the material orientation of the floor; the direction specified becomes the 1-direction of the material, the other in-plane axis becomes the 2-direction, while the 3-direction are always defined as the out-of-plane direction. It is not necessary to define the material orientation if an isotropic material is chosen.

Note that it is required to include a foundation slab at the the first floor (level 0) in order for the script to run properly. In Figure A.18 this is represented with the Concrete Slab. The material and thickness of this floor is not of importance to the properties of the building, as it is mainly included in order to tie the bottom of the walls to prevent spurious local modes.

A.2.13 Shell Connections

The "Shell Connections" sheet (Figure A.19) is used for setting the properties of the connection zones for the exterior walls, and any connection zones that may be defined in the "Floor Sections" sheet.

				Damping (Optional)				
		Fraction of...		Rayleigh Damping		Other Damping		
Name (Original)	Field Width	Thickness	Material	Alpha	Beta	Composite	Structural	Connect to
Walls	0,45	0,10	Wall Material					Floors/Beams only
Timber Floors	0,25	0,10	Timber Floor Material					NA

Figure A.19: Shell connections input

The first row of input is dedicated to the connection zones of the exterior walls. This line should not be removed, and the name should not be changed. The rest of the rows are for the floors with connector zones. The connector zones are created previously in the "Floor Sections" sheets, while the properties are set in this sheet. Choose a floor (with connector zones already created) from the drop-down menu in the column called "Name (Original)". In the "Field Width" column the width of the connector zone (see Figure A.20) is set, and the thickness of the zone is set as a fraction of the original floor/wall thickness in the next column. Choose the material from the drop-down menu, the material must be defined in the "Materials" sheet before it appears in the list. Next the user can define damping parameters, but this is optional. Note that any damping defined in the "Materials" sheet does not apply to the connection zones, only the damping from this sheet is included. Also note that using the composite damping option in combination with the other damping options may lead to issues, see subsection A.2.18 and the Abaqus documentation for more information. The final column "Connect to" does only apply to the exterior walls, and gives the choice of connecting the wall panels along the horizontal edges (floors/beams) only, or along all edges (floors/beams and columns).

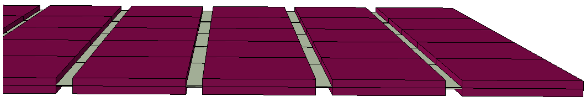


Figure A.20: Width of connection zone

A.2.14 Floor to Shaft Connections

The principle behind the connections between the floor and the shafts are described in section 3.6. The input sheet are shown in Figure A.21. First start by choosing the name of the floor that should be assigned a connection zone in the drop-down menu that appears when clicking a cell in the "Floor Name" column, all the floors defined in the "Floor Sections" sheet should be in the menu (if not it can be entered manually). It is crucial that the name of floor written/chosen in the "Floor Name" column is identical to floor defined in the "Floor Sections" sheet. Note that the tie between the floor and shaft is established for all floors, the input in this sheet only controls whether or not a connection zone with different properties from the original floor is created.

In the "Field Width" column the width of the connector zone (see Figure 3.11) is set, and the thickness of the zone is set as a fraction of the original floor thickness in the next column. The material of the connection zone is chosen from a drop-down menu in the "Material" column. The final four columns are used for setting the damping of the material in the connector zone. Note that any damping defined in the "Materials" sheet does not apply to the connection zones, only the damping from this sheet is included. Also note that using the composite damping option in combination with the other damping options may lead to issues, see subsection A.2.18 and the Abaqus documentation for more information.

Floor-to-shaft connection

Only modify if shaft is to be connected to the rest of the building

				Damping (Optional)			
		Fraction of Originial Section		Rayleigh Damping		Other Damping	
Floor Name	Field Width	Thickness	Material	Alpha	Beta	Composite	Structural
Concrete Floors	0,30	0,0500	Concrete				
Timber Floors	0,25	0,0500	Timber Floor Material				

Figure A.21: Floor-to-shaft connections input

A.2.15 Boundary Conditions

The spring stiffness of the foundation springs are set in the "Boundary Conditions" sheet. The values inserted in "Spring Stiffness" and "Dashpot Coeffcient" columns are the values per spring/dashpot (in general one set of springs/dashpots per column, in addition to one set at each shaft corner). The degrees of freedom (DOFs) follow the global axis system.

DOF	Description	Spring Stiffness	Dashpot Coefficient
1	Horizontal (x)	1,00E+09	0
2	Vertical (y)	1,00E+09	0
3	Horizontal (z)	1,00E+09	0
4	Rotation 1	0	0
5	Rotation 2	0	0
6	Rotation 3	0	0

Figure A.22: Boundary conditions input

A.2.16 Distributed/Point Mass

Additional non-structural distributed mass can be applied in the "Distributed Mass" sheet (Figure A.23). The mass is applied to all floors between "Start Level" and "End Level", including the start and end levels. There is no limit to how many masses that can be assigned to each level. The start and end levels is specified using the level indices from the "Grid (Y)" sheet.

Mass Per Area

Name	Start Level	End Level	Distributed Mass [kg/m2]
DistMass1	1	6	100
DistMass2	7	10	50
DistMass3	11	16	200

Figure A.23: Distributed mass input

Concentrated mass can be added to the intersections of the grid. The vertices where the point masses are applied are found by a imaginary box covering the volume between the start point and the end point. The start and end points are defined by the indices of the grid lines. Note that only intersections between the grid lines in the original grid system defined in the "Grid (XZ)" and "Grid (Y)" sheets are assigned point masses, intersections between members created using the "Add to Frame" sheet are omitted. Choose the part masses should be applied to in the pull-down menu, and set the magnitude per point mass.

Point Mass								
		Start Indices			End Indices			Magnitude
Name	Part	x	y	z	x	y	z	[kg]
Balcony Center	Frame	4	11	1	4	15	4	2500
Balcony Outer 1	Frame	4	11	0	4	16	0	1250
Balcony Outer 2	Frame	4	11	5	4	16	5	1250

Figure A.24: Point mass input

A.2.17 Wind (Eurocode)

The input in this sheet is to great extent directly from Eurocode 1 part 1-4 [23], and the method used for calculating the loads, including the formulas, are described further in subsection 2.3.3 of the thesis.

The first input is self-explanatory, simply choose the wind direction from the pull-down menu. The axis definition is the same as in the rest of the model.

The next section of input are parameters related to the structure, see Figure A.25

- The "Logarithmic Decrement (Structural)" input has two different methods. If "Abaqus Based" is chosen from the drop-down menu, the logarithmic decrement are determined from a free vibration step in the Abaqus analysis. The other method that can be chosen from the drop-down menu are "Specify", in this case the logarithmic decrement inserted in the "Value" column is used instead of the value from Abaqus.
- The input to the "Logarithmic Decrement (Aerodynamic)" is similar to the previous input. However instead of the "Abaqus Based" option there is a "Eurocode" option where Equation F.16 from the Eurocode is used.
- "First Nat. Freq. (In wind dir.)" has three options. "Abaqus Based" determines the frequency based on the aforementioned free vibration step, the option "Eurocode" uses rough estimate given in the Eurocode appendix ($f_1 =$

$46/h$), while the final option "Specify" allows the user to input the frequency directly in the "Value" column. Note that specifying the frequency does not change frequency of the FEA-model, only the input to the wind calculations.

- For the "Mode Shape Exponent" the only option currently implemented is to specify the value directly in the "Value" column. The mode shape is calculated by the estimate given by Equation F.13 in the Eurocode, with the specified value as the exponent in the expression.
- Finally the corner radius of the building can be specified. For most buildings $r = 0$ is appropriate.

Structural Input			
Parameter	**Value**	**Type**	**Description**
Logaritmic Decrement (Structural)	0,010	Abaqus Based	Value is ignored if type is "Abaqus Based"
Logaritmic Decrement (Aerodynamic)	0,000	Eurocode	Value is ignored if type is "Eurocode"
First Nat. Freq. (in wind dir.)	0,400	Abaqus Based	Value is ignored if type is "Abaqus Based" or "Eurocode"
Mode Shape Exponent	1,500	Specify	See Eq. F.13
Corner Radius	0,000	NA	Corner radius of building

Figure A.25: Structural input

The terrain category and reference wind speed ($V_{b,0}$) are exactly the same inputs as described in the Eurocode. The script allows to base the calculations of the load and the acceleration on two different return periods to accommodate for difference guidelines, the return period and probability factor are discussed in section 2.3.3 of the thesis. The next section of input parameters are related to the wind speed and the turbulence. The value is usually 1.0, but recommendations and rules for all the parameters are given in the Eurocode clause given in the "Ref." column. The final input is the "Sample height for acceleration results" where the height coordinate of the floor where the acceleration should be evaluated is specified.

A.2.18 Analysis Parameters

The first part (Figure A.26) of the sheet called "Analysis Parameters" is for setting the finite element size and type used for meshing the different parts of the structure. Fill in the maximum element size in the "Element Size" column and choose the element type from the drop-down menu. Note that some element sizes and/or types may lead to errors for certain types of analyses.

Mesh		
Part Name	**Element Size**	**Element Type**
Frame	1	B32
Walls	1	S4R
Floors	1	S4R
Shafts	1	S4R

Figure A.26: Mesh settings

The next part (Figure A.27) of the sheet is dedicated to the setup of the analysis steps. The first four steps are a part of the first analysis job called "TTBJob" while the final step listed are run as a part of the second analysis job called "WindJob". The second part (and the free vibration step of the first) of the analysis is only a part of the *TTB_3D_EC_wind.py* script and is ignored when running the script *TTB_3D.py*.

- The first step is a static step where gravity is applied, more loads can be added manually in Abaqus after generating the model or by modifying the script.
- The second step is a frequency step used to extract the natural frequencies and mode shapes of the structure. Set the number of modes to be calculated in the column called "Number of Modes". There is also the option to use or not use the SIM architecture in the calculation. The SIM option has consequences on the damping of the structure. The Rayleigh and the structural damping specified on material and element level in the previous sheets are compatible with the SIM based architecture turned on. If the SIM based calculation is turned off, only global/modal damping and the composite damping (which is a combination of local and global damping) are considered. For more information the user are referred to section 3.12 and subsection A.2.19 of the thesis, in addition to the Abaqus docs, especially the section "Damping in a linear dynamic analysis".
- The third step is called "Free Vibration" and is a modal dynamics step. The purpose of this step is to determine the logarithmic decrement of the building, caused by the different damping settings defined in the previous sheets. An impulse load is applied at the top of the building in the wind direction specified in the wind-load sheet. The building is then allowed to freely vibrate. The logarithmic decrement is calculated later in the script based on the magnitude of the peaks. The free vibration step is also used to determine the first natural frequency (in the wind direction) for the wind calculations. Set the length of each time step in the "Time-Step" column and the total duration in the "Duration" column. It's important that the time-steps are small enough to capture the peaks and the duration long enough to allow at least 4-5 full cycles. Note that for the results from the free vibration step

to be valid the first mode in the direction specified needs to be a bending mode and at least the 3-4 first vibration cycles needs to be like a sine wave with only one peak per cycle. The displacements used in the calculations are sampled at the center of the top floor of the building (ref. chapter 7 of the thesis).

- The fourth step is also a modal dynamics step. In this step a dynamic pressure load is applied to one of the walls, the load amplitude is defined in a .txt named `load_amplitude.txt` file located in the same folder as the scripts. The amplitude included in the digital appendix is a random example, and may be exchanged with a similar file with the same name and location. This step may be appropriate to use for e.g. analyzing the response of the structure to a specific time-history of wind loading. Set the length of each time step in the "Time-Step" column and the total duration in the "Duration" column.
- The final step is a static step used to calculate the response (deflection) of the structure to wind load according to the method given in Eurocode 1 - part 1.4 [23]. See subsection 2.3.3 of the thesis and subsection A.2.17 for more information on the wind calculations.

Each step can be included or omitted as the user wants, but it should be noted that the "Free Vibration" and "Modal Dynamics" steps must be proceeded by the "Frequency" step. Also the "Static (EC Wind)" step relies on data calculated based on the results from the "Free Vibration" step.

Steps

Step Type	Include	Number of modes	Time Step	Duration	SIM-Based Dynamics	Description
Static	0	-	-	-	-	Static analysis (TTBJob)
Frequency	1	50	-	-	1	Frequency/mode shape
Free Vibration	1	-	0,1	25	-	Determine logarithmic decrement for use in wind
Modal Dynamics	0	-	0,1	100	-	Modal dynamics analysis
Static (EC Wind)	1	-	-	-	-	Wind calculation based on

Figure A.27: Step settings

The final group of settings (Figure A.28) is related to setting up and running the analysis jobs. The first job, "TTBJob", is the main job and is used for all types of analyses. The second job, "WindJob", is ran in combination with "TTBJob" when analysing the response of the structure to wind load according to the Eurocode. Note that for all the parameters and results for EC wind loading to be calculated properly, both jobs needs to be created and "Auto Run" must be turned on. However, for e.g. a simple frequency extraction it is not necessary to create the "WindJob" and "Auto Run" for "TTBJob" is optional. It is also possible to specify the number of CPUs to be used in the analyses.

Job				
Job Name	Create	Auto Run	Number of CPUS	Description
TTBJob	1	1	2	Analysis of a TTB
WindJob	1	0	2	Wind Caluculations based on Eurocode

Figure A.28: Job settings

A.2.19 Step Level Damping

Damping can also be applied at a global level for each of the modal dynamics steps (including the free vibration step, who is a modal dynamics step). The first row of tables belongs to the free vibration step, while the second row is for the step named "Modal Dynamics Step".

The first global damping method that can be assigned to the steps are "Direct modal" damping. Here a "Critical Damping Factor" (i.e. damping ratio) can be assigned to one or more modes. Enter the start and end mode in the two first columns respectively, and the critical damping factor (as a percentage) in the final column of the first table (Figure A.29).

Direct modal		
Start Mode	End Mode	Critical Damping Factor
1	3	2,000 %

Figure A.29: Direct modal damping input

The second option for adding damping at the step level is "Composite Modal" damping. Here all the composite damping values previously specified for each material, connector section etc. are converted into mass weighted damping ratios for the modes in the range defined by the specified start and end mode in the table shown in Figure A.30.

Composite Modal	
Start Mode	End Mode
1	3

Figure A.30: Composite modal damping input

The final way of defining global damping is by using "Rayleigh Damping", which is simply a linear combination of the global stiffness and mass matrices (see subsection 2.2.3 of the thesis for more information on Rayleigh damping). Specify the start and end modes, as well as the α and β (also denoted as α_0 and α_1) factors in the table (Figure A.31).

Rayleigh			
Start Mode	End Mode	Alpha	Beta
1	1	0,001	0,002

Figure A.31: Rayleigh damping input

A.3 Running the Script

After OpenPyXl (subsection A.1.1) is installed, the paths to the files and folders (subsection A.1.2) are updated and the input file is finished and saved (section A.2), the program is ready to be used. Start by choosing either the *TTB_3D.py* script for a "normal" analysis or the *TTB_3D_EC_wind.py* script for running an analysis where the wind load is calculated and applied. *TTB_3D_EC_wind.py* can also be used if any of the results from the free vibration step is of interest. There are at least two different methods to run the chosen script:

A.3.1 Running the Script from the GUI

Open Abaqus CAE and click the "Run Script" button highlighted in Figure A.32.

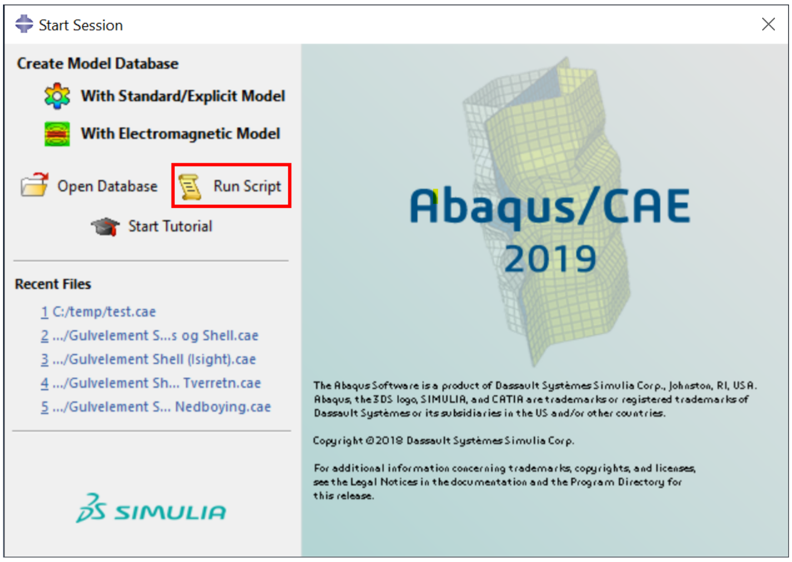


Figure A.32: Running the script from the GUI - Step 1

Then locate the script (*TTB_3D.py* or *TTB_3D_EC_wind.py*) in the window that appears (Figure A.33) and click "OK", the script starts automatically. The script may take a few minutes to complete, depending on the complexity of the model and the number of analysis steps defined.

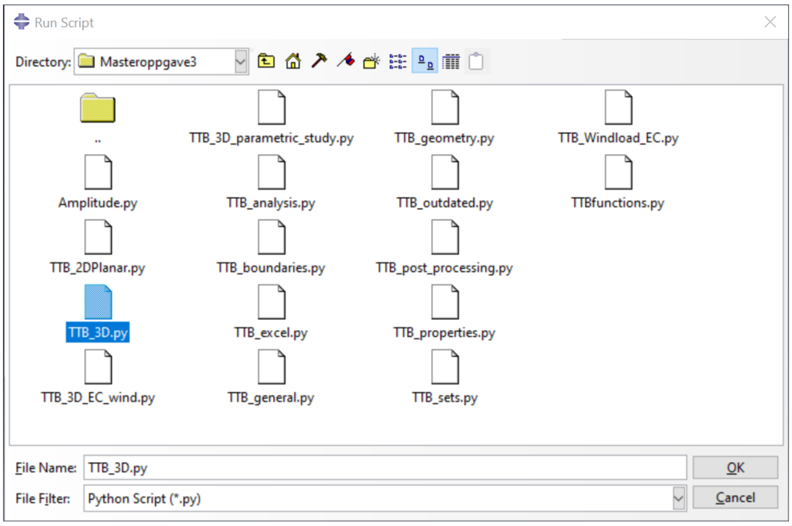


Figure A.33: Running the script from the GUI - Step 2

While the script is running information is written to the message area (Figure A.34. These messages might be useful to ensure that the model works as it should, or to help to resolve any errors that might occur.

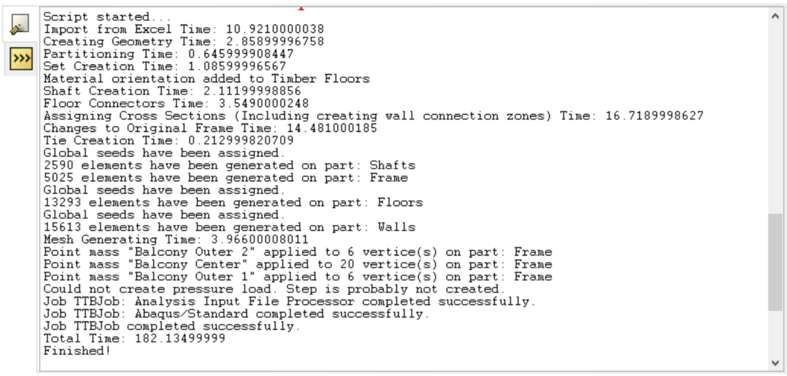


Figure A.34: Typical output to message area when running *TTB_3D.py*

A.3.2 Running the Script from the Command Line (CMD)

The script can also be initialized through the command line by opening "CMD" and typing the following command:

```
abaqus cae script="filepath.py"
```

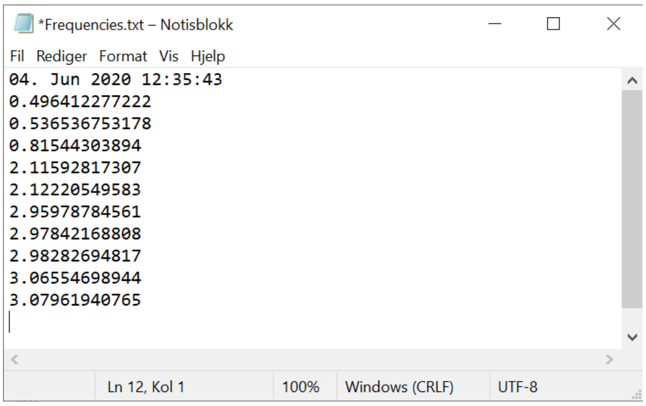
where "filepath.py" is replaced with the path to the Python-script. This will open the GUI and run the script, hence the method is equivalent to running the script trough the GUI. Alternatively the script can be ran without the GUI by typing the following command:

```
abaqus cae noGUI="filepath.py"
```

However its strongly to recommended to open the GUI, at least the first time a new configuration is tested, and check visually that the model is generated properly.

A.3.3 Result Files

If the "Auto Run" option is turned on in the "Analysis Parameters" sheet of the input file, the script also does some post-processing of the results and writes the results to the Abaqus working directory specified in the script (see subsection A.1.2). For the *TTB_3D.py* script the only result file created are *Frequencies.txt* (Figure A.35), a file containing all the frequencies calculated in the frequency step.


*Frequencies.txt – Notisblokk
Fil Rediger Format Vis Hjelp

```
04. Jun 2020 12:35:43
0.496412277222
0.536536753178
0.81544303894
2.11592817307
2.12220549583
2.95978784561
2.97842168808
2.98282694817
3.06554698944
3.07961940765
```

Ln 12, Kol 1 | 100% | Windows (CRLF) | UTF-8

Figure A.35: Frequencies.txt

When running the *TTB_3D_EC_wind.py* script three additional files are created: *FreeVibrationResults.txt* (Figure A.36), a file containing the results of the free vibration step, *EurocodeWindAccelerationResults.txt* (Figure A.37), a file containing the results of acceleration calculation based on the Eurocode [23] and finally *WindCalculationParameters.txt* (Figure A.38) which contains most of the parameters used in the wind-related calculations.

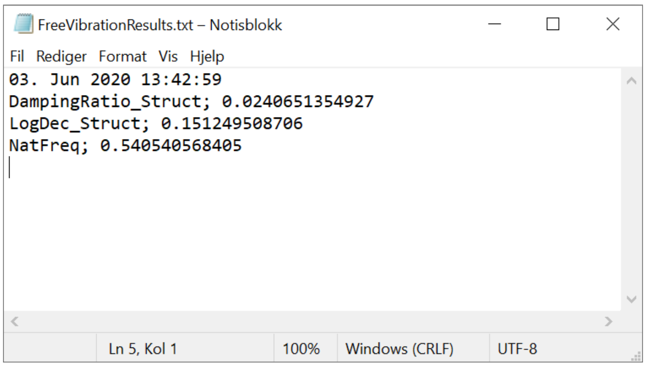


Figure A.36: FreeVibrationResults.txt

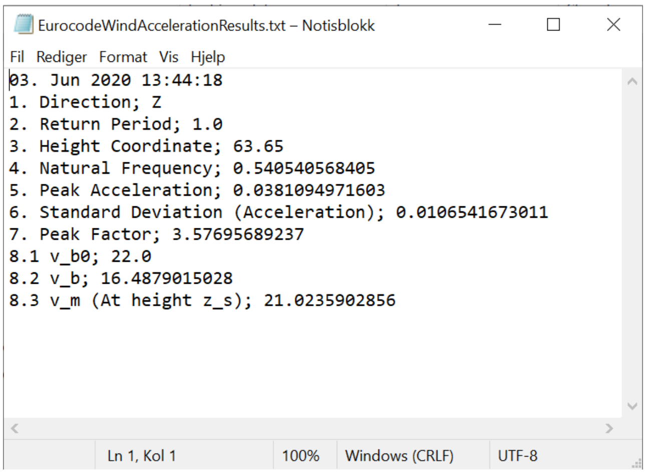
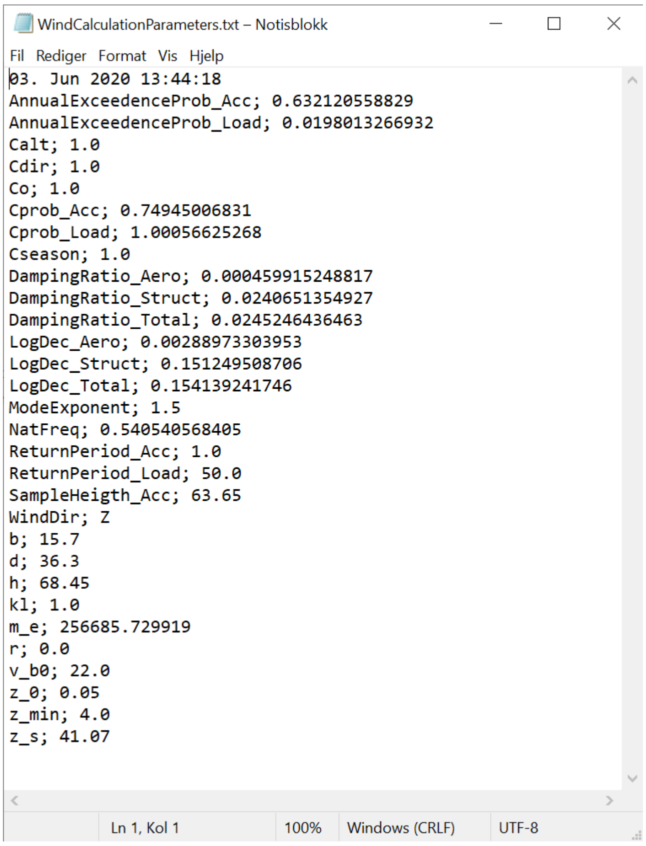


Figure A.37: EurocodeWindAccelerationResults.txt



WindCalculationParameters.txt – Notisblokk

Fil Rediger Format Vis Hjelp

```
03. Jun 2020 13:44:18
AnnualExceedenceProb_Acc; 0.632120558829
AnnualExceedenceProb_Load; 0.0198013266932
Calt; 1.0
Cdir; 1.0
Co; 1.0
Cprob_Acc; 0.74945006831
Cprob_Load; 1.00056625268
Cseason; 1.0
DampingRatio_Aero; 0.000459915248817
DampingRatio_Struct; 0.0240651354927
DampingRatio_Total; 0.0245246436463
LogDec_Aero; 0.00288973303953
LogDec_Struct; 0.151249508706
LogDec_Total; 0.154139241746
ModeExponent; 1.5
NatFreq; 0.540540568405
ReturnPeriod_Acc; 1.0
ReturnPeriod_Load; 50.0
SampleHeigth_Acc; 63.65
WindDir; Z
b; 15.7
d; 36.3
h; 68.45
kl; 1.0
m_e; 256685.729919
r; 0.0
v_b0; 22.0
z_0; 0.05
z_min; 4.0
z_s; 41.07
```

Ln 1, Kol 1 100% Windows (CRLF) UTF-8

Figure A.38: WindCalculationParameters.txt

A.4 Isight

Simulia Isight [37] is a great tool for running multiple analyses automatically. This guide shows how to setup a sensitivity study, like the one performed in chapter 5 of this thesis, and a model updating routine similar to the one in chapter 6. Once the application "Isight Design Gateway" is opened, a empty model should be initialized like in Figure A.39.

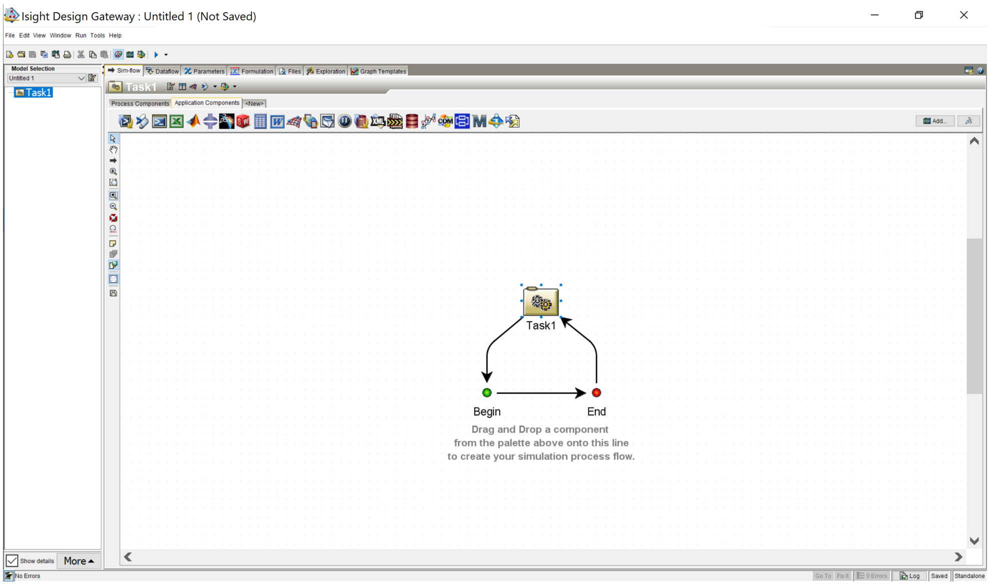


Figure A.39: An empty Isight model

To be able to use Isight the "Auto Run" setting in "Analysis Parameters" sheet of the input file needs to be activated, (ref. subsection A.2.18). In addition the script should be tested and the model should be checked visually in the GUI of CAE before running Isight, as it is much more difficult to notice and diagnose errors trough Isight.

A.4.1 Adding the Application Components

The first step is to add the application components to the simulation flow. Click the "Application Components" tab highlighted with a red box in Figure A.40, then drag and drop the "Excel" and "Simcode" components into the simulation flow. Another useful tool might be the "Calculator" which can be used for e.g. simple pre- or post-processing of different parameters.

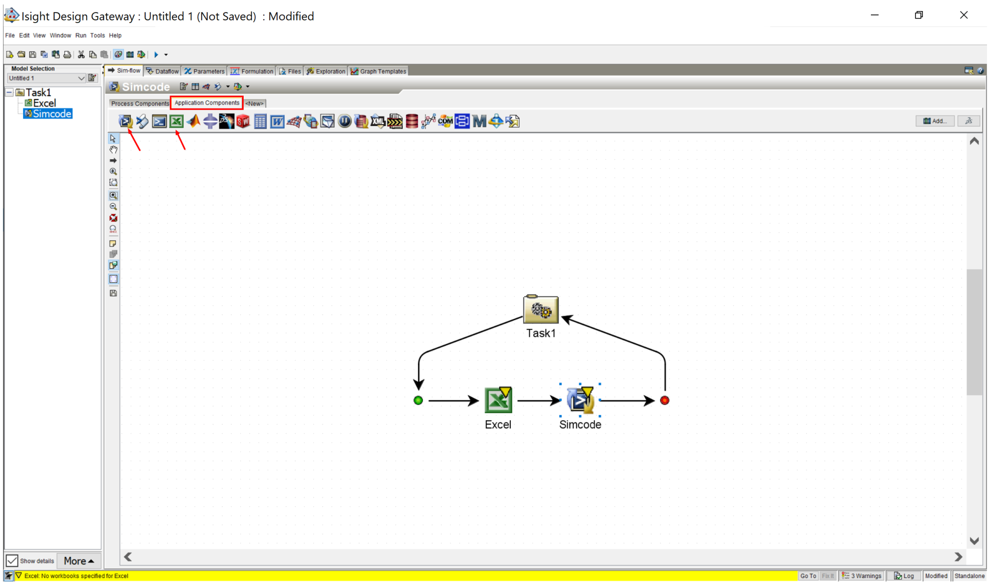


Figure A.40: Adding the application components

A.4.2 Excel Component Setup

Bring up the "Component Editor" by double-clicking the Excel icon in the simulation flow. Then follow the steps in the list below, illustrated in Figure A.41:

1. Click "Browse", locate and open the Excel input file.
2. Pick a cell that contains a parameter value that is to be changed by Isight.
3. Give the parameter a name.
4. Click the red "+" to save the parameter, the parameter should appear in the list below and the cell should turn dark yellow/gold.

Repeat steps 2-4 for all desired parameters. If many parameters are to be updated simultaneously with the same value, it can be time saving to do so by introducing formulas (such that many cells are updated based on a single cell) in the Excel sheet before loading it in Isight and map the reference cell only.

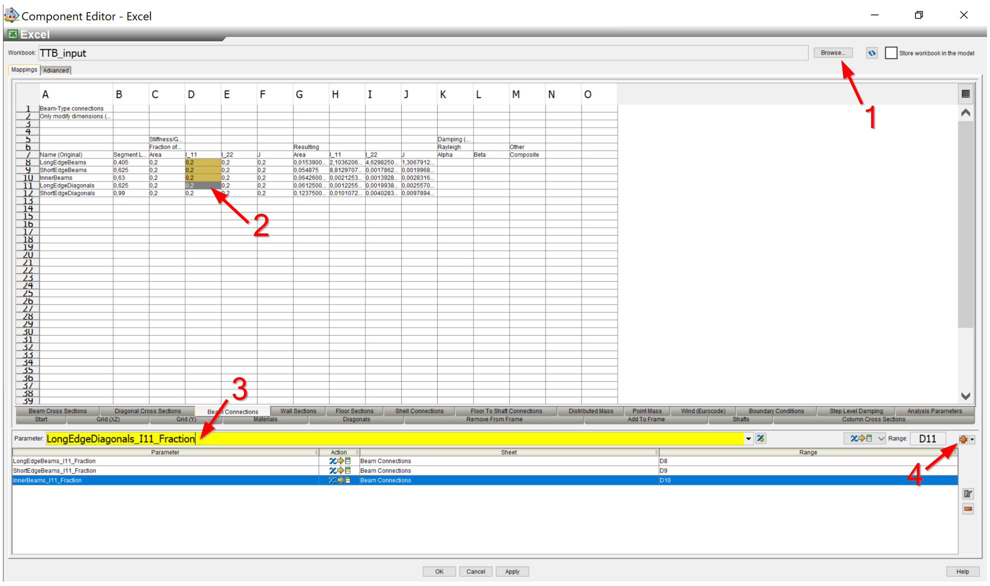


Figure A.41: Excel component setup - Step 1

After the steps in Figure A.41 are completed, click "Apply" and move on to the steps in Figure A.42:

1. Open the advanced settings.
2. Check the box "Save Excel file after execution". Make sure that the path is exactly the same as the path to the input file specified in the script (subsection A.1.2).

The setup of the Excel component is completed, click "Apply" and "OK" to save and close the settings.

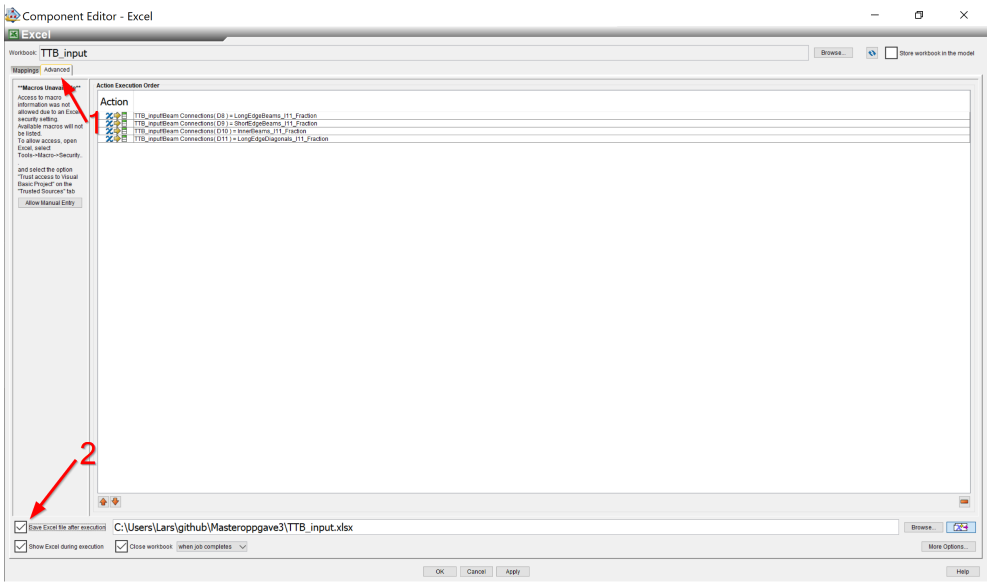


Figure A.42: Excel component setup - Step 2

A.4.3 Simcode Component Setup

Double-click the Simcode icon to open the "Component Editor". Part 1 of the setup is listed below and illustrated in Figure A.43:

1. Set the script type to "Windows Batch"
2. Type the following command in the editor: `abaqus cae noGUI="script_path.py"`, replace *script_path.py* with the path to the main script that will be used to perform the analysis.

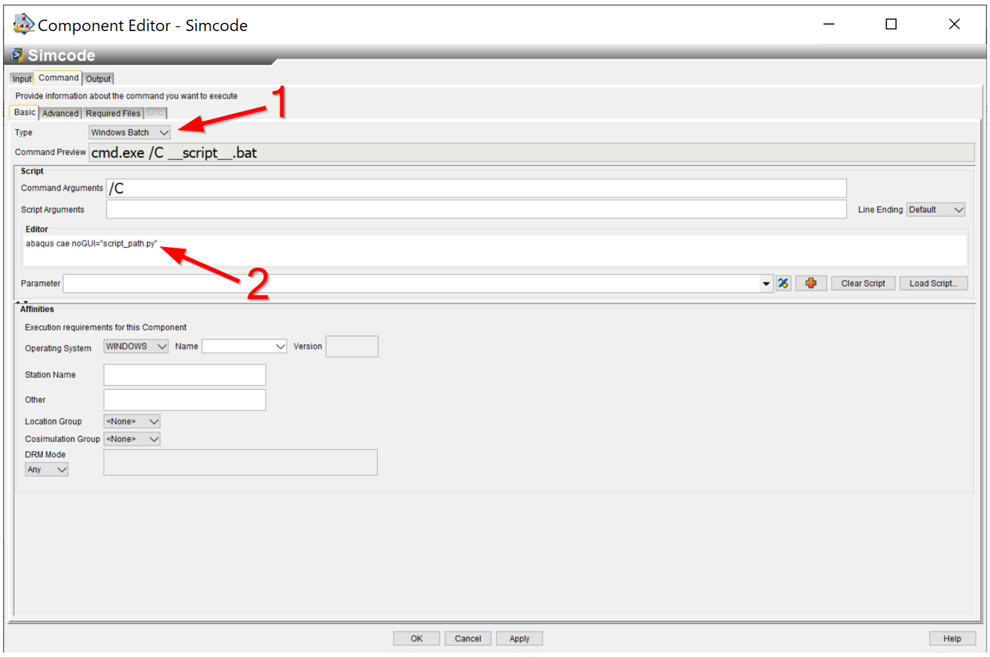


Figure A.43: Simcode component setup - Step 1

Then move on to Figure A.44:

1. Navigate to the "Advanced" tab.
2. Uncheck the option "There is output to the Standard Error stream".
3. Increase the number of seconds in the "Execution takes longer than..." option. The number of seconds must be higher than the duration of a single iteration. Alternatively the option can be unchecked.

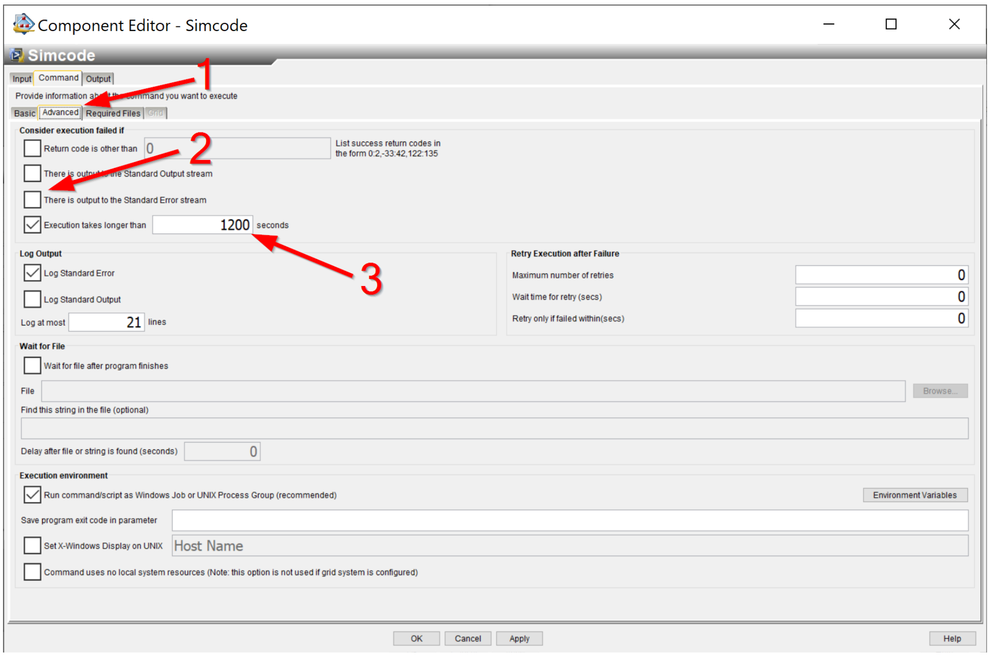


Figure A.44: Simcode component setup - Step 2

Part 3 (Figure A.45) of the settings is about reading the results from the output file(s).

1. Navigate to the "Output" tab. Click the on the box in the center of the window to define a new data source.
2. Browse to locate the results file. The script must be ran at least once outside Isigth before setting up Isight in order to have a result file to use as a template in the setup. The values inside the files used for the setup are irrelevant, but the structure of it needs to be correct.
3. Choose "General Text" (depending on the structure of the result file) from the "Format" menu and press "OK".

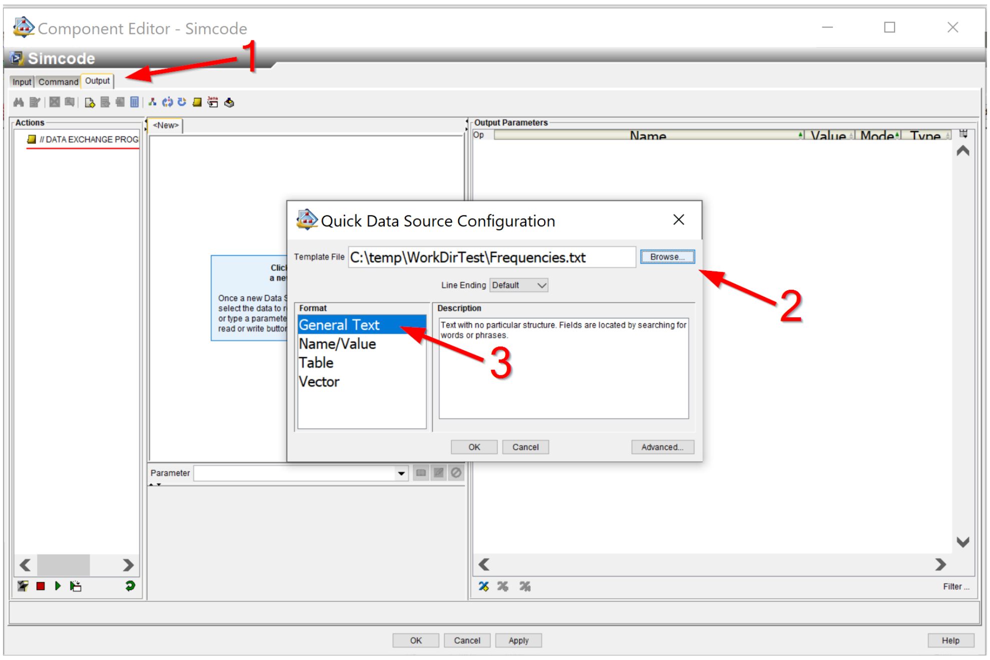


Figure A.45: Simcode component setup - Step 3

The process of mapping the results to parameters are shown in Figure A.46:

1. Give the parameter a name by typing in yellow box.
2. Specify the line number and, if relevant, the word number of the result to be mapped to the parameter.
3. Press the book icon next to the yellow box to add the mapping.
4. Check that the parameter is mapped in the "Output Parameters" list on the right hand side.

Repeat the sequence above for all results of interest, and click apply when finished.

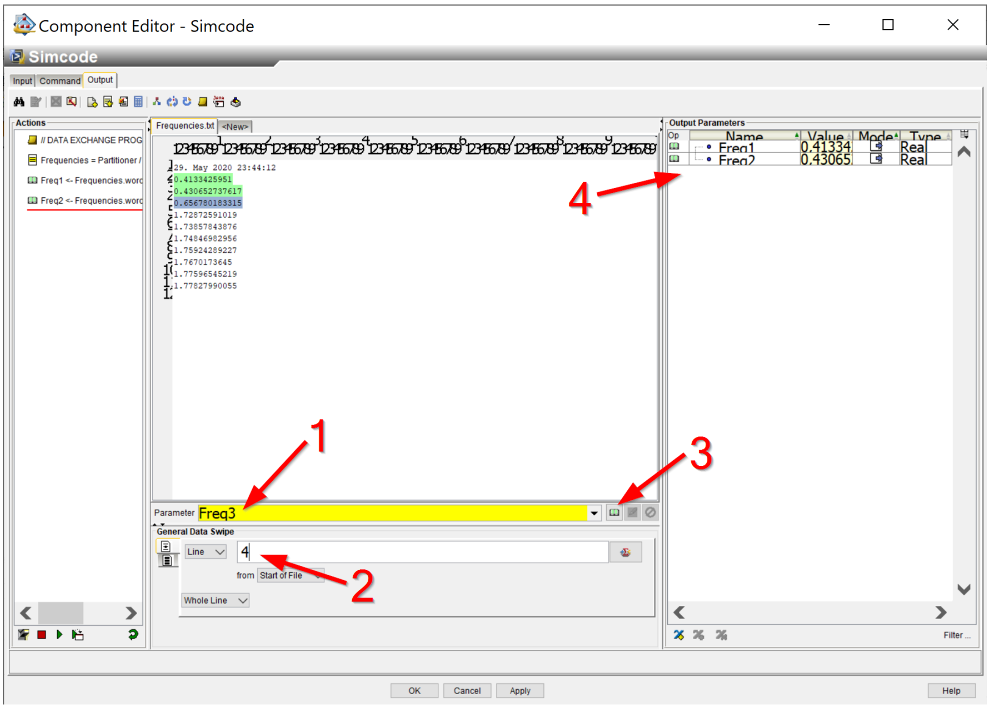


Figure A.46: Simcode component setup - Step 4

Finally setup how the result file(s) is stored. The procedure is shown in Figure A.47.

1. Navigate back to the "Command" tab.
2. Click "Required Files".
3. Press the button marked with "...".
4. Choose the "Absolute Path" option from the pull-down menu.
5. Browse to find the results file previously loaded.
6. Make sure that the path in the "Path" field is the same as the path to working directory specified inside the script (ref. subsection A.1.2).
7. Finally check the "None" option inside the "Destination" box.

If results from more than one output file are used, repeat the procedure for each file.

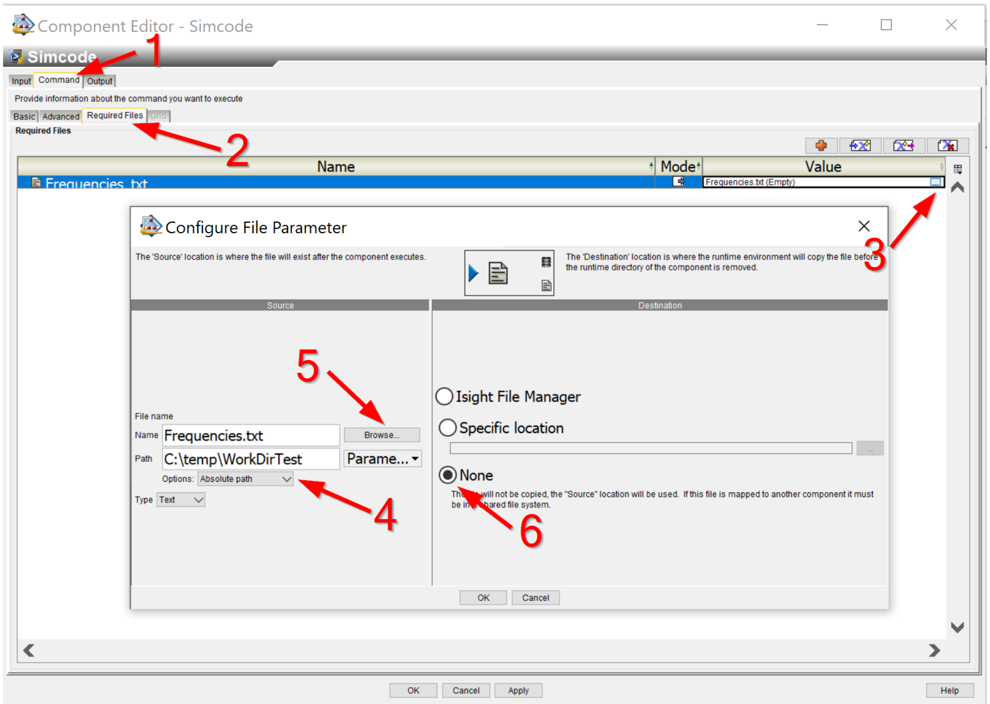


Figure A.47: Simcode component setup - Step 5

The setup of the Simcode component is now complete, press "Apply" and "OK" to save the settings and exit the component editor.

A.4.4 Adding a Process Component

When the setup of the application components is completed, it is time to add the "Process Component". The "Process Component" is responsible for updating the input parameters and keeping track of the output provided by the applications in the sim-flow. There are many process components designed for different applications included in Isight. Subsection A.4.5 describes how to setup a parameter study, while the "Target Solver" is described in subsection A.4.6.

The "Process Components" are located under the tab with the same name (highlighted in Figure A.48). Drag the selected component into the sim-flow at the position highlighted in the figure, on top of the existing component. Click "OK" in the box that appears.

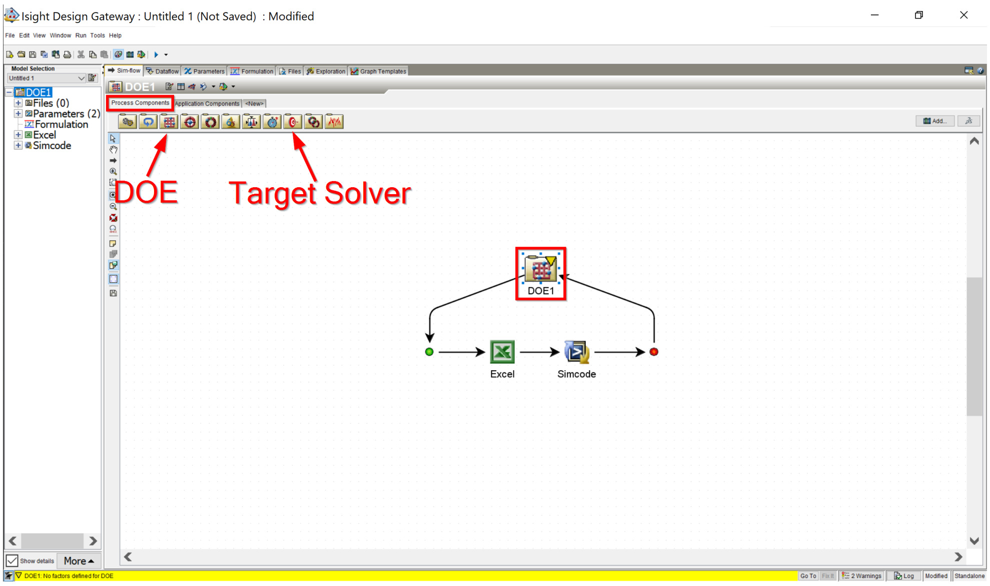


Figure A.48: Adding a process component

A.4.5 Parameter Study (DOE) Configuration

The sensitivity study in chapter 5 of the thesis is performed using the parameter study functionality of Isight. The parameter study is a part of the DOE (Design of experiments) component. After the component is placed in the sim-flow, open the component editor by double-clicking the icon.

Choose the "Parameter Study" as the "DOE Technique" from the drop-down menu, as shown in Figure A.49.

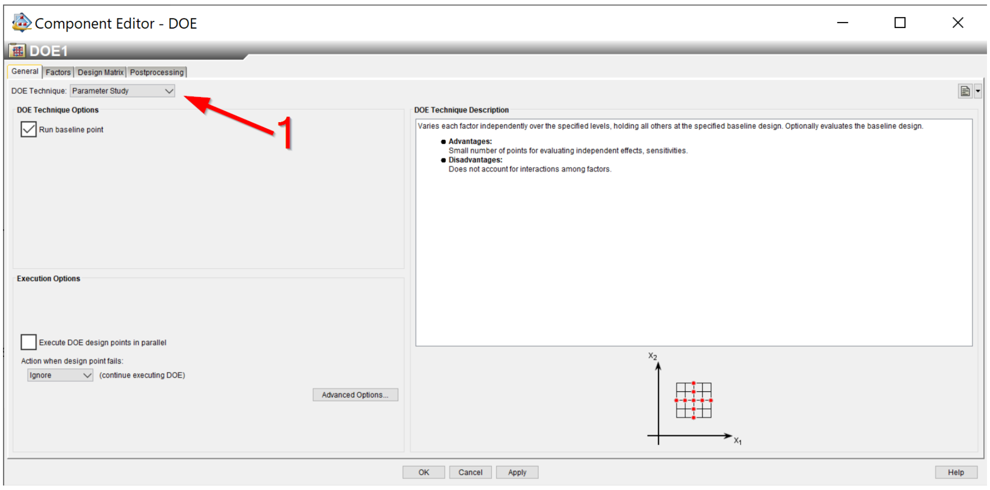


Figure A.49: DOE component setup - Part 1

Then set the settings of the input parameters. The procedure is shown in Figure A.50, and listed below:

1. The settings are accessed by choosing the "Factors" tab.
2. Check the boxes for all the parameters to be included in the parameter study.
3. Set the "Relation" type. This specifies if the values should be defined as absolute values, percentages of the starting-value etc.
4. Set the number of different values to be tested for each variable.
5. Set the range of the values to be tested. Remember that the upper and lower limits depends on "Relation" set in step 3. All the parameter values to be tested can be seen in the "Values" column to the far right.

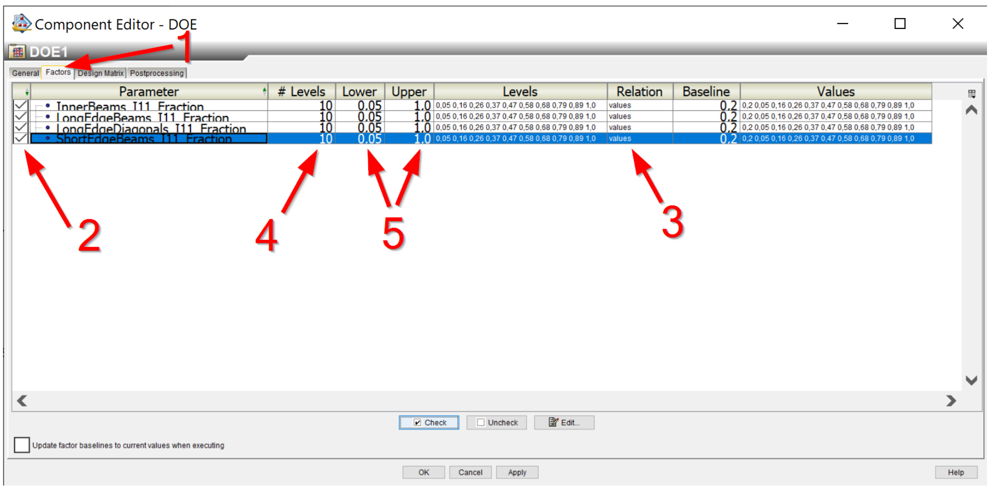


Figure A.50: DOE component setup - Part 2

The output parameters that is used for measuring the sensitivities of the parameters are picked in the next step. The procedure is shown in Figure A.50, and listed below:

1. Go to the "Postprocessing" tab.
2. Check the boxes for all the output parameters to be included.

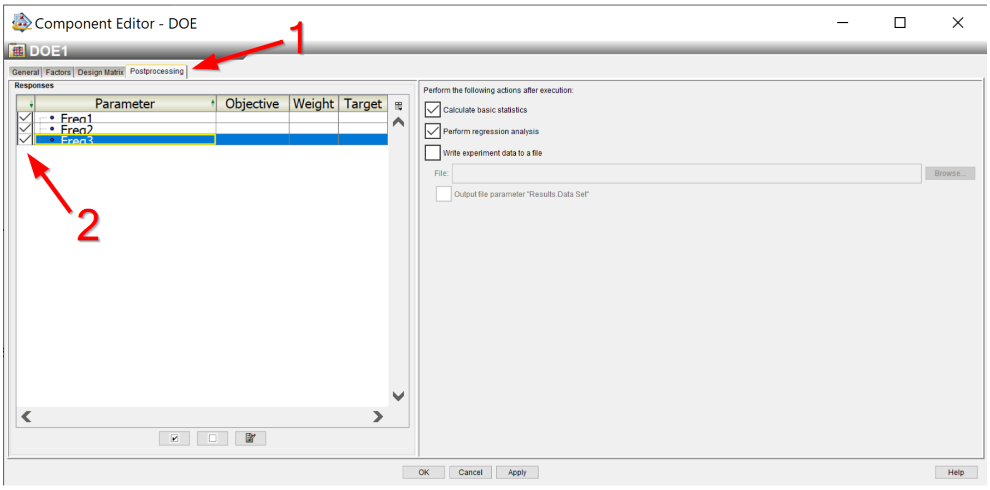


Figure A.51: DOE component setup - Part 3

The setup of the DOE component is now complete, press "Apply" and "OK" to save the settings and exit the component editor. The analysis can be started by pressing "Run Model" from the "Run" menu in the toolbar. All open Abaqus and Excel instances should be closed before running the analysis. It is advised to follow the first few iterations closely to make sure that both the input and output is updated correctly.

A.4.6 Target Solver Configuration

The model updating performed in chapter 6 is performed using the "Target Solver" component. Add the "Target Solver" as the process component of the sim-flow, as described in subsection A.4.4, and open the component editor.

First set the settings for the target variables as demonstrated in Figure A.52:

1. Check the box for all the result variables to be used in the analysis.
2. Set the target value for each variable. The target is often results of physical experiments.

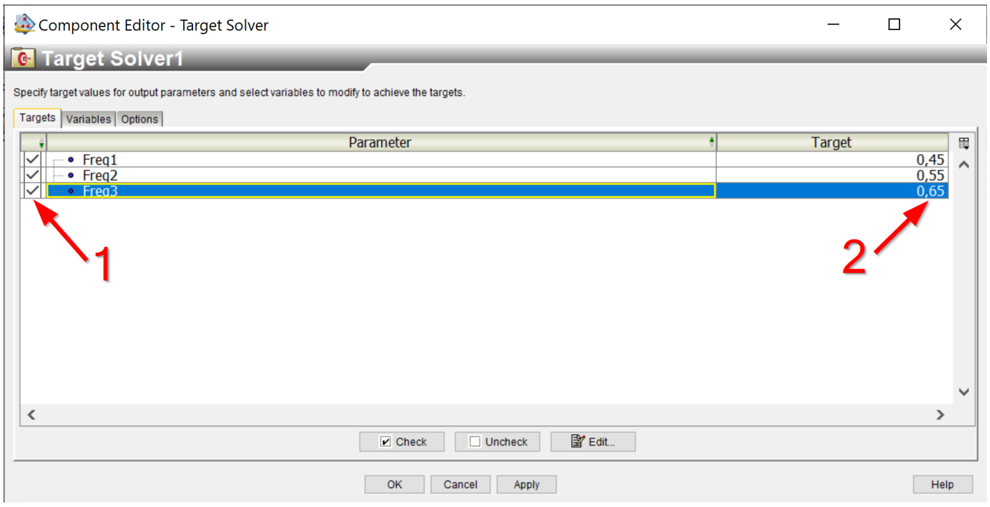


Figure A.52: Target Solver setup - Part 1

Next set the variables to be updated by the target solver (Figure A.53)

1. Navigate to the "Variables" tab.
2. Check the box next to all the variables that are to be updated in the process.
3. Set the lower and upper limits of the variables.

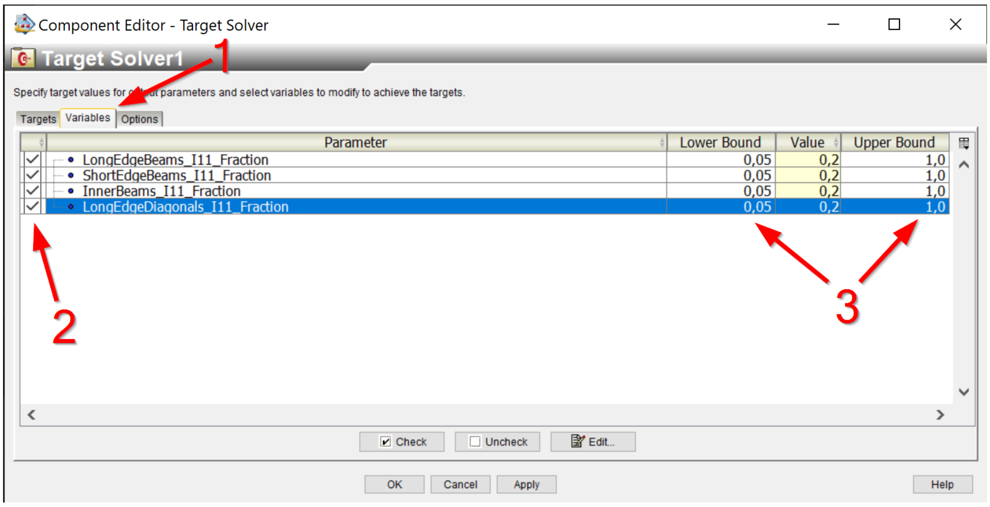


Figure A.53: Target Solver setup - Part 2

Finally set the analysis options (Figure A.54)

1. Go to the options tab.
2. Set the maximum number of iterations.

3. Set the target tolerance. The lower the tolerance, the more iterations are needed to reach the target. A good starting point for the tolerance could be the margin of error of the experimental results used as target values.

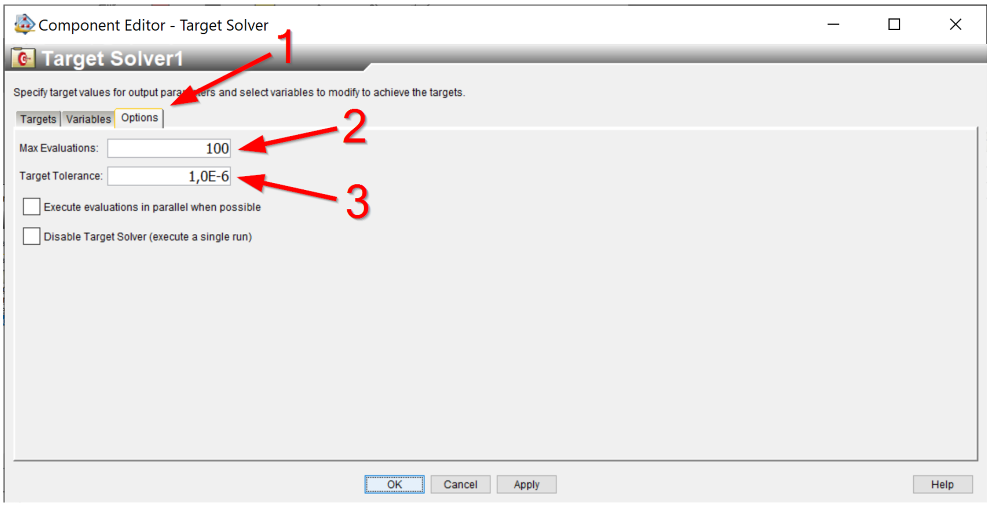


Figure A.54: Target Solver setup - Part 3

The setup of the "Target Solver" component is now completed, press "Apply" and "OK" to save the settings and exit the component editor. The Isight process can be started by pressing "Run Model" from the "Run" menu in the toolbar. All open Abaqus and Excel instances should be closed before running the analysis. It is advised to follow the first few iterations closely to make sure that both the input and output is updated correctly.

Appendix B

Digital Appendix

A digital appendix containing various files relevenat to the thesis is delivered directly to prof. Malo at the Department of Structural Engineering at NTNU. The following files and folders are included:

- **Test Results:** This folder contains all results of the sensitivity study, model updating and wind analyses.
- **Calculations, Estimates etc.:** This folder contains the files related to the estimation of the floor stiffness, and some additional calculations for e.g. the non-structural mass as well as a Mathcad-sheet for calculating Rayleigh coefficients.
- **Parametric Model**: This folder contains all the scripts needed for the use of the parametric model. The input sheet is also located here. Finally, the input file used for the base model is also located here.
- **User Guide**: The user guide from Appendix A.

Appendix C

Python Scripts

This appendix includes all the scripts necessary for running the parametric model. Each script is preceded by a short explanation. All the files are also included in the digital appendix. The scripts included are:

- **TTB_3D.py:** This file is the main script for creating and analysing the parametric model. This script gathers and uses functions written in the other files of this thesis. Note: This script does not include the wind analysis, use the script *TTB_3D_EC_wind.py* instead for wind analysis according to Eurocode.
- **TTB_3D_EC_Wind.py:** This file is the main script for creating and analysing the parametric model including wind loads according to the rules provided in the Eurocode. Apart from the wind calculations the script is identical to TTB_3D.py.
- **TTB_analysis.py:** This file contains all the functions related to setting up and running the analysis. Examples include adding loads and non-strucural mass, creating steps, generate mesh and setting up the job.
- **TTB_boundaries.py:** This file contains all the functions related to boundary conditions and interaction between the different parts of the structure.
- **TTB_excel.py:** This file contains all the functions related to importing the input data from the input file generated in Excel. The data from Excel is mainly imported as dictionaries and lists to allow for further usage of the data inside python.
- **TTB_general.py:** This file contains basic functions for e.g. initializing the model and creating parts.
- **TTB_geometry.py:** This file contains all the functions related to generating the geometry of the building. Beams, columns, bracing, walls, floors etc. are created using the functions from this script.
- **TTB_post_processing.py:** This file contains functions used to gather and

process the results after a simulation. Used for getting the eigenfrequencies, calculating the damping ratio/logarithmic decrements and writing the results to a .txt file.

- **TTB_properties.py:** This file contains functions used to assign different properties to objects. Such properties include material data and cross sections.
- **TTB_sets.py:** This file contains all the functions related to creating sets of all kinds of objects in Abaqus e.g. beams, columns, surfaces etc...
- **TTB_Windload_EC.py:** This file contains all the functions and formulas for calculating the wind load according to Eurocode 1.

C.1 TTB_3D.py

This file is the main script for creating and analysing the parametric model. This script gathers and uses functions written in the other files of this thesis. Note: This script does not include the wind analysis, use the script *TTB_3D_EC_wind.py* instead for wind analysis according to Eurocode.

```python
# This is the main file for the 3D model of a TTB, including wind analysis.
# -------------- Input file/folder paths -------------
# All the locations specified must exist (i.e folders must be created BEFORE running the script)
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'
# Path to the Excel-file containing the input:
inputFile = 'C:\\Users\\username\\TTBParametricModel\\TTB_input.xlsx'
# Path to Abaqus working directory (all result files will be stored here):
workDir = 'C:\\temp'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sys
import sketch
import part
import material
import section
import assembly
import material
import mesh
import time
import odbAccess
import load
import random
import step
import os

sys.path.append(scriptsFolder)
```

```python
os.chdir(workDir)

## Custom functions
from TTB_analysis import *
from TTB_excel import *
from TTB_boundaries import *
from TTB_general import *
from TTB_geometry import *
from TTB_post_processing import *
from TTB_properties import *
from TTB_sets import *

start_time = time.time()
print('\nScript started...')
session.viewports['Viewport: 1'].setValues(displayedObject=None)
close_odbs()

# Write negative values to file to indicate error.
# (The negative values are updated with the correct ones if the run
↪  is successful)
errorLst = [-1]*50
write_to_file(errorLst, 'Frequencies.txt', 'w+')

# -------------- Imports -------------
z_coord_lst, x_coord_matrix = xz_grid_from_xlsx('Grid (XZ)',
↪  wb_name=inputFile)
y_coord_lst = y_grid_from_xlsx('Grid (Y)', wb_name=inputFile)
grid = [x_coord_matrix, y_coord_lst, z_coord_lst]

shaft_dict = shaft_dict_from_xlsx('Shafts', wb_name=inputFile)

materials_dict = create_material_dict_from_xlsx('Materials',
↪  wb_name=inputFile)

damping_dict = damping_dict_from_xlsx('Materials',
↪  wb_name=inputFile)

diag_dict = diagonals_dict_from_xlsx('Diagonals', wb_name=inputFile)

remove_dict = remove_dict_from_xlsx('Remove From Frame',
↪  wb_name=inputFile)

add_dicts  = add_to_frame_from_xlsx('Add To Frame',
↪  wb_name=inputFile)
```

```
connector_dict = create_connector_dict_from_xlsx2('Beam
↪  Connections', wb_name=inputFile)

crossSectionsCols = cross_section_dict_from_xlsx('Column Cross
↪  Sections', wb_name=inputFile)

orientationsCols = {'CornerColumns': (0, 0, -1),
                    'LongEdgeColumns': (-1, 0, 0),
                    'ShortEdgeColumns': (0, 0, -1),
                    'InnerColumns': (-1, 0, 0)}

crossSectionsBeams = cross_section_dict_from_xlsx('Beam Cross
↪  Sections', wb_name=inputFile)

orientationsBeams = {'LongEdgeBeams': (-1, 0, 0),
                     'ShortEdgeBeams': (0, 0, -1),
                     'InnerBeams': (0, 0, -1)}

crossSectionsDiags = cross_section_dict_from_xlsx('Diagonal Cross
↪  Sections', wb_name=inputFile)

orientationsDiags = {'LongEdgeDiagonals': (-1, 0, 0),
                     'ShortEdgeDiagonals': (0, 0, -1)}

floor_dict = floor_dict_from_xlsx('Floor Sections',
↪  wb_name=inputFile)

sectionsWalls = shell_section_dict_from_xlsx('Wall Sections',
↪  wb_name=inputFile)

shell_connector_dict = create_shell_connector_dict_from_xlsx('Shell
↪  Connections', wb_name=inputFile)

bc_dict = create_boundary_spring_dict_from_xlsx('Boundary
↪  Conditions', wb_name=inputFile)

mass_dict = mass_dict_from_xlsx('Distributed Mass',
↪  wb_name=inputFile)

point_mass_dict = point_mass_dict_from_xlsx("Point Mass",
↪  wb_name=inputFile)

```

```
mesh_dict = mesh_dict_from_xlsx('Analysis Parameters', wb_name=inputFile)

ec_wind_dict = ec_wind_param_from_xlsx('Wind (Eurocode)', wb_name=inputFile)

floor_to_shaft_dict = floor_shaft_connection_from_xlsx('Floor To Shaft Connections', wb_name=inputFile)

print('Import from Excel Time: ' + str(time.time()-start_time))

# -------------- Initialize Model and Parts -------------
change_model_name('Tall Timber Building')
TTBModel = get_model()

framePart = create_part(part_name='Frame')
floorPart = create_part(part_name='Floors')
wallPart = create_part(part_name='Walls')
shaftPart = create_part(part_name='Shafts')


# -------------- Material -------------
create_material_from_dict(materials_dict)
add_material_damping(damping_dict)
shell_connector_material(materials_dict, sectionsWalls, floor_dict, shell_connector_dict)
floor_to_shaft_material(materials_dict, floor_dict, floor_to_shaft_dict)


# --------------- Create Geometry ------------
geo_time = time.time()
build_frame(framePart, diag_dict, connector_dict, grid, floor_dict)
build_floors(floorPart, 0, 17, grid)
create_walls(wallPart, 0, 17, grid)
print('Creating Geometry Time: ' + str(time.time()-geo_time))


# --------------- Create Instances ------------
create_instance(framePart)
create_instance(floorPart)
create_instance(wallPart)
create_instance(shaftPart)
```

```

# --------------- Partition Shells ------------
parti_time = time.time()
partition_shells(floorPart, grid, XYPLANE)
partition_shells(wallPart, grid, XYPLANE)
partition_shells(wallPart, grid, XZPLANE)
partition_shells(wallPart, grid, YZPLANE)
print('Partitioning Time: ' + str(time.time()-parti_time))


# --------------- Create Sets ------------
set_time = time.time()
colSet, beamSet, diagSet = create_sets(framePart)
sets_of_cols(framePart, colSet, grid)
sets_of_beams(framePart, beamSet, grid)
sets_of_diagonals(framePart, diagSet, grid)

floorSet = create_set_all_floors(floorPart)
set_of_floor_types(floorPart, floor_dict, grid)
outer_floor_edges_set(floorPart,grid)
surface_of_bottom_floor(floorPart, grid)

wallSet = create_set_all_walls(wallPart)
longWall1Set = set_of_selected_walls(wallPart, wallSet, 0, 'yz')
wall_surfaces(wallPart)
print('Set Creation Time: ' + str(time.time()-set_time))


# --------------- Orient Floors ------------
orient_floors(floorPart, floor_dict)


# --------------- Create Shaft Geometry and Sets ------------
shaft_time = time.time()
create_shafts(shaftPart, floorPart, framePart, shaft_dict, grid)
sets_of_shaft_floor_edges(floorPart, shaft_dict, grid)
set_of_all_shafts(shaftPart)
set_of_single_shaft(shaftPart, shaft_dict, grid)
shaft_edges_for_wall_ties(shaftPart, shaft_dict, grid)
print('Shaft Creation Time: ' + str(time.time()-set_time))


# --------------- Create connector elements for floors
 ↪  ---------------
```

```
floor_time = time.time()
floor_connector_partition(floorPart, floor_dict, shell_connector_dict, grid)
set_of_floor_connectors(floorPart, floor_dict, shell_connector_dict, grid)
floor_shaft_partition(floorPart, floor_dict, shaft_dict, floor_to_shaft_dict, grid)
floor_to_shaft_set(floorPart, floor_dict, shaft_dict, floor_to_shaft_dict, grid)
print('Floor Connectors Time: ' + str(time.time()-floor_time))


# --------------- Assign Cross Sections ------------
cs_time = time.time()
section_assignment(framePart, crossSectionsCols, orientationsCols)
section_assignment(framePart, crossSectionsBeams, orientationsBeams)
section_assignment(framePart, crossSectionsDiags, orientationsDiags)
#
walls_with_connectors_section_assignment_auto(wallPart, sectionsWalls, shell_connector_dict, grid)
#
floor_assignment_from_dict(floorPart, floorSet, floor_dict, grid)
floor_connector_assignment(floorPart, floor_dict, shell_connector_dict)
assign_floor_shaft_connector(floorPart, floor_dict, floor_to_shaft_dict)
shaft_section_assignment(shaftPart, sectionsWalls)

connector_assignment_auto_generalized_profile(framePart, crossSectionsBeams, connector_dict, materials_dict)
connector_assignment_auto_generalized_profile(framePart, crossSectionsDiags, connector_dict, materials_dict)
print('Assigning Cross Sections (Including creating wall connection zones) Time: ' + str(time.time()-cs_time))


# ---------- Alternate Original Frame -------------
alter_time = time.time()
remove_wires(framePart, remove_dict, grid)
add_wires(framePart, add_dicts)
colSet, beamSet, diagSet = create_sets(framePart)
sets_of_cols(framePart, colSet, grid)
sets_of_beams(framePart, beamSet, grid)
sets_of_added_wires(framePart, add_dicts)
```

```
section_assignment(framePart, add_dicts['Section'],
 ↪  add_dicts['Orientation'])
assign_connector_added_wire(framePart, add_dicts, materials_dict)
print('Changes to Original Frame Time: ' +
 ↪  str(time.time()-alter_time))


# --------------- Establish Ties ------------
tie_time = time.time()
assembly_regenerate()
edges_for_wall_ties_set(shaftPart, framePart, floorPart,
 ↪  shell_connector_dict)
floor_surfaces(floorPart)
set_of_bottom_nodes(framePart, grid)
tie_floors_node_to_surf(floorPart, framePart)
wall_ties(wallPart)
shaft_floor_tie(shaftPart, floorPart, shaft_dict)
column_to_slab_tie(floorPart, framePart)
print('Tie Creation Time: ' + str(time.time()-tie_time))


# --------------- Meshing ------------
mesh_time = time.time()
create_mesh_auto(mesh_dict)
print('Mesh Generating Time: ' + str(time.time()-mesh_time))


# --------------- Add BC Springs ------------
create_boundary_springs_from_dict(framePart, bc_dict)
create_boundary_springs_from_dict(shaftPart, bc_dict)


# --------------- Create Steps ------------
steps_from_xlsx('Analysis Parameters', wb_name=inputFile)


# --------------- Add Step-Level Damping ------------
step_damping_from_xlsx('Step Level Damping', wb_name=inputFile)


# --------------- Loads/mass ------------
mass_from_dict(floorPart, floorSet, mass_dict, grid)

point_mass_from_dict(point_mass_dict, grid)
```

```python

try:
    add_gravity('StaticStep')
except:
    print('Could not add gravity. Step is probably not created.')

try:
    amplitude_from_file(scriptsFolder+'\\load_amplitude.txt')
    create_pressure('WindLoad', wallPart, longWall1Set, 500, 'ImportedAmplitude', 'ModalDynamicsStep')
except:
    print('Could not create pressure load. Step is probably not created.')

# --------------- Create Set Of Output Node ------------
create_output_node_set(floorPart, grid)

# --------------- Regenerate Assembly ------------
assembly_regenerate()


# -------------- Create and run job -------------
try:
    TTBModel.steps['FreeVibrationStep'].suppress()
except:
    pass
try:
    TTBModel.steps['Static_Wind_Eurocode'].suppress()
except:
    pass

run_boolean = job_from_xlsx('Analysis Parameters', row_nr=21, wb_name=inputFile)


# -------------- Post Processing -------------
if run_boolean:
    freqs = get_eigenfreqs()
    write_to_file(freqs, 'Frequencies.txt', 'w+')

end_time = time.time()
print('Total Time: '+ str(end_time-start_time))
print('Finished!')
```

C.2 TTB_3D_EC_wind.py

This file is the main script for creating and analysing the parametric model including wind loads according to the rules provided in the Eurocode. Apart from the wind calculations the script is identical to TTB_3D.py.

```
# This is the main file for the 3D model of a TTB, including wind analysis.
# -------------- Input file/folder paths -------------
# All the locations specified must exist (i.e folders must be created BEFORE running the script)
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'
# Path to the Excel-file containing the input:
inputFile = 'C:\\Users\\username\\TTBParametricModel\\TTB_input.xlsx'
# Path to Abaqus working directory (all result files will be stored here):
workDir = 'C:\\temp'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sys
import sketch
import part
import material
import section
import assembly
import material
import mesh
import time
import odbAccess
import load
import random
import os
import step

sys.path.append(scriptsFolder)
os.chdir(workDir)
```

```python

## Custom functions
from TTB_analysis import *
from TTB_excel import *
from TTB_boundaries import *
from TTB_general import *
from TTB_geometry import *
from TTB_post_processing import *
from TTB_properties import *
from TTB_sets import *
from TTB_Windload_EC import *

start_time = time.time()
print('\nScript started...')
session.viewports['Viewport: 1'].setValues(displayedObject=None)
close_odbs()

# Write negative values to file to indicate error.
# (The negative values are updated with the correct ones if the run
↪ is successful)
errorLst = [-1]*50
write_to_file(errorLst, 'Frequencies.txt', 'w+')
write_to_file(errorLst, 'FreeVibrationResults.txt', 'w+')
write_to_file(errorLst, 'EurocodeWindAccelerationResults.txt', 'w+')
write_to_file(errorLst, 'WindCalculationParameters.txt', 'w+')

# -------------- Imports -------------
z_coord_lst, x_coord_matrix = xz_grid_from_xlsx('Grid (XZ)',
↪ wb_name=inputFile)
y_coord_lst = y_grid_from_xlsx('Grid (Y)', wb_name=inputFile)
grid = [x_coord_matrix, y_coord_lst, z_coord_lst]

shaft_dict = shaft_dict_from_xlsx('Shafts', wb_name=inputFile)

materials_dict = create_material_dict_from_xlsx('Materials',
↪ wb_name=inputFile)

damping_dict = damping_dict_from_xlsx('Materials',
↪ wb_name=inputFile)

remove_dict = remove_dict_from_xlsx('Remove From Frame',
↪ wb_name=inputFile)

```

```
add_dicts  = add_to_frame_from_xlsx('Add To Frame',
↪  wb_name=inputFile)

diag_dict = diagonals_dict_from_xlsx('Diagonals', wb_name=inputFile)

connector_dict = create_connector_dict_from_xlsx2('Beam
↪  Connections', wb_name=inputFile)

crossSectionsCols = cross_section_dict_from_xlsx('Column Cross
↪  Sections', wb_name=inputFile)

orientationsCols = {'CornerColumns': (0, 0, -1),
                    'LongEdgeColumns': (-1, 0, 0),
                    'ShortEdgeColumns': (0, 0, -1),
                    'InnerColumns': (-1, 0, 0)}

crossSectionsBeams = cross_section_dict_from_xlsx('Beam Cross
↪  Sections', wb_name=inputFile)

orientationsBeams = {'LongEdgeBeams': (-1, 0, 0),
                     'ShortEdgeBeams': (0, 0, -1),
                     'InnerBeams': (0, 0, -1)}

crossSectionsDiags = cross_section_dict_from_xlsx('Diagonal Cross
↪  Sections', wb_name=inputFile)

orientationsDiags = {'LongEdgeDiagonals': (-1, 0, 0),
                     'ShortEdgeDiagonals': (0, 0, -1)}

floor_dict = floor_dict_from_xlsx('Floor Sections',
↪  wb_name=inputFile)

sectionsWalls = shell_section_dict_from_xlsx('Wall Sections',
↪  wb_name=inputFile)

shell_connector_dict = create_shell_connector_dict_from_xlsx('Shell
↪  Connections', wb_name=inputFile)

bc_dict = create_boundary_spring_dict_from_xlsx('Boundary
↪  Conditions', wb_name=inputFile)

mass_dict = mass_dict_from_xlsx('Distributed Mass',
↪  wb_name=inputFile)

```

```
point_mass_dict = point_mass_dict_from_xlsx('Point Mass', wb_name=inputFile)

mesh_dict = mesh_dict_from_xlsx('Analysis Parameters', wb_name=inputFile)

floor_to_shaft_dict = floor_shaft_connection_from_xlsx('Floor To Shaft Connections', wb_name=inputFile)

ec_wind_dict_xlxs = ec_wind_param_from_xlsx('Wind (Eurocode)', wb_name=inputFile)

print('Import from Excel Time: ' + str(time.time()-start_time))

# -------------- Initialize Model and Parts -------------
change_model_name('Tall Timber Building')
TTBModel = get_model()

framePart = create_part(part_name='Frame')
floorPart = create_part(part_name='Floors')
wallPart = create_part(part_name='Walls')
shaftPart = create_part(part_name='Shafts')


# -------------- Material -------------
create_material_from_dict(materials_dict)
add_material_damping(damping_dict)
shell_connector_material(materials_dict, sectionsWalls, floor_dict, shell_connector_dict)
floor_to_shaft_material(materials_dict, floor_dict, floor_to_shaft_dict)

# --------------- Create Geometry ------------
geo_time = time.time()
build_frame(framePart, diag_dict, connector_dict, grid, floor_dict)
build_floors(floorPart, 0, 17, grid)
create_walls(wallPart, 0, 17, grid)
print('Creating Geometry Time: ' + str(time.time()-geo_time))


# --------------- Create Instances ------------
create_instance(framePart)
create_instance(floorPart)
create_instance(wallPart)
```

```python
create_instance(shaftPart)


# --------------- Partition Shells ------------
parti_time = time.time()
partition_shells(floorPart, grid, XYPLANE)
partition_shells(wallPart, grid, XYPLANE)
partition_shells(wallPart, grid, XZPLANE)
partition_shells(wallPart, grid, YZPLANE)
print('Partitioning Time: ' + str(time.time()-parti_time))


# --------------- Create Sets ------------
set_time = time.time()
colSet, beamSet, diagSet = create_sets(framePart)
sets_of_cols(framePart, colSet, grid)
sets_of_beams(framePart, beamSet, grid)
sets_of_diagonals(framePart, diagSet, grid)

floorSet = create_set_all_floors(floorPart)
set_of_floor_types(floorPart, floor_dict, grid)
outer_floor_edges_set(floorPart,grid)
surface_of_bottom_floor(floorPart, grid)

wallSet = create_set_all_walls(wallPart)
longWall1Set = set_of_selected_walls(wallPart, wallSet, 0, 'yz')
wall_surfaces(wallPart)
print('Set Creation Time: ' + str(time.time()-set_time))

# --------------- Orient Floors ------------
orient_floors(floorPart, floor_dict)

# --------------- Create Shaft Geometry and Sets ------------
shaft_time = time.time()
create_shafts(shaftPart, floorPart, framePart, shaft_dict, grid)
sets_of_shaft_floor_edges(floorPart, shaft_dict, grid)
set_of_all_shafts(shaftPart)
set_of_single_shaft(shaftPart,shaft_dict, grid)
shaft_edges_for_wall_ties(shaftPart, shaft_dict, grid)
print('Shaft Creation Time: ' + str(time.time()-set_time))

# --------------- Create connector elements for floors
    ---------------
floor_time = time.time()
```

```
floor_connector_partition(floorPart, floor_dict, shell_connector_dict, grid)
set_of_floor_connectors(floorPart, floor_dict, shell_connector_dict, grid)
floor_shaft_partition(floorPart, floor_dict, shaft_dict, floor_to_shaft_dict, grid)
floor_to_shaft_set(floorPart, floor_dict, shaft_dict, floor_to_shaft_dict, grid)
print('Floor Connectors Time: ' + str(time.time()-floor_time))

# --------------- Assign Cross Sections ------------
cs_time = time.time()
section_assignment(framePart, crossSectionsCols, orientationsCols)
section_assignment(framePart, crossSectionsBeams, orientationsBeams)
section_assignment(framePart, crossSectionsDiags, orientationsDiags)
#
walls_with_connectors_section_assignment_auto(wallPart, sectionsWalls, shell_connector_dict, grid)
#
floor_assignment_from_dict(floorPart, floorSet, floor_dict, grid)
floor_connector_assignment(floorPart, floor_dict, shell_connector_dict)
assign_floor_shaft_connector(floorPart, floor_dict, floor_to_shaft_dict)
shaft_section_assignment(shaftPart, sectionsWalls)

connector_assignment_auto_generalized_profile(framePart, crossSectionsBeams, connector_dict, materials_dict)
connector_assignment_auto_generalized_profile(framePart, crossSectionsDiags, connector_dict, materials_dict)

print('Assigning Cross Sections (Including creating wall connection zones) Time: ' + str(time.time()-cs_time))

# ---------- Alternate Original Frame -------------
alter_time = time.time()
remove_wires(framePart, remove_dict, grid)
add_wires(framePart, add_dicts)
colSet, beamSet, diagSet = create_sets(framePart)
sets_of_cols(framePart, colSet, grid)
sets_of_beams(framePart, beamSet, grid)
sets_of_added_wires(framePart, add_dicts)
section_assignment(framePart, add_dicts['Section'], add_dicts['Orientation'])
```

```python
assign_connector_added_wire(framePart, add_dicts, materials_dict)
print('Changes to Original Frame Time: ' + str(time.time()-alter_time))

# --------------- Establish Ties ------------
tie_time = time.time()
assembly_regenerate()
edges_for_wall_ties_set(shaftPart, framePart, floorPart, shell_connector_dict)
floor_surfaces(floorPart)
set_of_bottom_nodes(framePart, grid)
tie_floors_node_to_surf(floorPart, framePart)
wall_ties(wallPart)
shaft_floor_tie(shaftPart, floorPart, shaft_dict)
column_to_slab_tie(floorPart, framePart)
print('Tie Creation Time: ' + str(time.time()-tie_time))

# --------------- Meshing ------------
mesh_time = time.time()
create_mesh_auto(mesh_dict)
print('Mesh Generating Time: ' + str(time.time()-mesh_time))


# --------------- Add BC Springs ------------
create_boundary_springs_from_dict(framePart, bc_dict)
create_boundary_springs_from_dict(shaftPart, bc_dict)


# --------------- Create Steps ------------
steps_from_xlsx('Analysis Parameters', wb_name=inputFile)


# --------------- Add Step-Level damping ------------
step_damping_from_xlsx('Step Level Damping', wb_name=inputFile)


# --------------- Loads/mass ------------
mass_from_dict(floorPart, floorSet, mass_dict, grid)

point_mass_from_dict(point_mass_dict, grid)

try:
    add_gravity('StaticStep')
except:
```

```
    print('Could not add gravity. Step is probably not created.')

try:
    free_vibration_impulse(ec_wind_dict_xlxs, floorPart, grid)
except:
    print('Could not create free vibration impulse. Step is
     ↪  probably not created.')

try:
    amplitude_from_file(scriptsFolder+'\\load_amplitude.txt')
    create_pressure('WindLoad', wallPart, longWall1Set, 500,
     ↪  'ImportedAmplitude', 'ModalDynamicsStep')
except:
    print('Could not create pressure load. Step is probably not
     ↪  created.')


# --------------- Create Set Of Output Node ------------
create_output_node_set(floorPart, grid)


# --------------- Regenerate Assembly ------------
assembly_regenerate()


# -------------- Create and run job -------------
try:
    TTBModel.steps['Static_Wind_Eurocode'].suppress()
except:
    pass

run_boolean_TTBJob = job_from_xlsx('Analysis Parameters',
 ↪  row_nr=21, wb_name=inputFile)

try:
    TTBModel.steps['Static_Wind_Eurocode'].resume()
except:
    pass


# -------------- Post Processing -------------
if run_boolean_TTBJob:
    free_vib_res = free_vib_res_dict(ec_wind_dict_xlxs, floorPart)
    freqs = get_eigenfreqs()
```

```python
    write_to_file(freqs, 'Frequencies.txt', 'w+')
    write_to_file(free_vib_res, 'FreeVibrationResults.txt', 'w+')

print('Part 1 Finished! Time: '+ str(time.time()-start_time))


# -------------- EC-wind -------------
pt2_time = time.time()
if run_boolean_TTBJob:
    param_dict = create_wind_param_dict(ec_wind_dict_xlxs, free_vib_res, grid)
    try:
        apply_EC_wind_force(framePart, colSet, grid, param_dict, adjust_to_grid=True)
    except:
        print('Could not apply Eurocode wind loading. Step is probably not created. Or try setting "adjust_to_grid" to False')
    try:
        TTBModel.steps['FreeVibrationStep'].suppress()
    except:
        pass
    try:
        TTBModel.steps['FrequencyStep'].suppress()
    except:
        pass
    run_boolean_WindJob = job_from_xlsx('Analysis Parameters', row_nr=22, wb_name=inputFile)

    acc_wind_res = acc_res_dict_EC(param_dict)
    write_to_file(acc_wind_res, 'EurocodeWindAccelerationResults.txt', 'w+')
    write_to_file(param_dict, 'WindCalculationParameters.txt', 'w+')

print('Part 2 Finished! Time: '+ str(time.time()-pt2_time))
print('Total Time: '+ str(time.time()-start_time))
```

C.3 TTB_analysis.py

This file contains all the functions related to setting up and running the analysis. Examples include adding loads and non-strucural mass, creating steps, generate mesh and setting up the job.

```python
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys
import step

sys.path.append(scriptsFolder)

from TTB_general import *
from TTB_sets import *

# -------------- Create mesh -------------
# This function meshes the specified part (or only the set if
↪   specified) with the selected element size and type.
def createMesh(part_, eleSize, eleName, subSet=None):
    eleConst = elemcode_string_to_constant(eleName)
    if subSet:
        set = subSet
    else:
        set = part_
```

```
    if len(set.faces) == 0:
        edgesForMesh = set.edges
        meshRegion = regionToolset.Region(edges=edgesForMesh)
        part_.seedEdgeBySize(edges=edgesForMesh, size=eleSize)
    else:
        facesForMesh = set.faces
        meshRegion = regionToolset.Region(faces=facesForMesh)
        part_.seedPart(size=eleSize)
        if ('tri' in eleName) or ('3' in eleName):
            part_.setMeshControls(elemShape=TRI, regions=facesForMesh)
        else:
            part_.setMeshControls(elemShape=QUAD_DOMINATED, regions=facesForMesh)
    element = mesh.ElemType(elemCode=eleConst, elemLibrary=STANDARD)
    part_.setElementType(regions=meshRegion, elemTypes=(element,))

# -------------- Create mesh by dictionary-------------
# This function meshes all the parts with names and properties specified in mesh_dict.
def create_mesh_auto(mesh_dict):
    model = get_model()
    for partName in mesh_dict.keys():
        try:
            part = model.parts[partName]
            eleSize, eleName = mesh_dict[partName]
            createMesh(part, eleSize, eleName)
            part.generateMesh()
        except:
            print('Could not mesh part: ' + str(partName))

# -------------- Convert element name string to abaqus constant -------------
# Converts a string with the elementcode to an Abaqus constant defining the chosen element type.
def elemcode_string_to_constant(elemcode_str):
    if elemcode_str.lower() == 'b31':
        return B31
    if elemcode_str.lower() == 'b31h':
        return B31H
    if elemcode_str.lower() == 'b32':
        return B32
    if elemcode_str.lower() == 'b33':
        return B33
```

```
    if elemcode_str.lower() == 'b33h':
        return B33H
    if elemcode_str.lower() == 't3d2':
        return T3D2
    if elemcode_str.lower() == 't3d2h':
        return T3D2H
    if elemcode_str.lower() == 'stri3':
        return STRI3
    if elemcode_str.lower() == 's3':
        return S3
    if elemcode_str.lower() == 's3r':
        return S3R
    if elemcode_str.lower() == 'stri65':
        return STRI65
    if elemcode_str.lower() == 's4':
        return S4
    if elemcode_str.lower() == 's4r':
        return S4R
    if elemcode_str.lower() == 's4r5':
        return S4R5
    if elemcode_str.lower() == 's8r':
        return S8R
    if elemcode_str.lower() == 's8r5':
        return S8R5
    if elemcode_str.lower() == 's9r5':
        return S9R5
    print('Unknown element code: '+ elemcode_str)

# -------------- Create Steps -------------
# Functions for creating different types of analysis steps
def create_static_step(name='StaticStep', prevStep='Initial',
 ↪  desc=''):
    model = get_model()
    model.StaticStep(name=name, previous=prevStep, description=desc)

def create_freq_step(name='FrequencyStep', nModes=30,
 ↪  prevStep='Initial', desc='', SIMBased=False):
    model = get_model()
    if SIMBased:
        model.FrequencyStep(name=name, numEigen=nModes,
         ↪  previous=prevStep, description=desc,
         ↪  simLinearDynamics=ON, normalization=MASS)
    else:
```

```python
        model.FrequencyStep(name=name, numEigen=nModes, previous=prevStep, description=desc, simLinearDynamics=OFF)

def create_modal_dyn_step(name='ModalDynamicsStep', prevStep='FrequencyStep', desc='', period=60, stepSize=0.1):
    model = get_model()
    model.ModalDynamicsStep(name=name, previous=prevStep, description=desc, timePeriod=period, incSize=stepSize)
    fieldOutputKeys = model.fieldOutputRequests.keys()
    fieldOutputKeys.sort()
    key = fieldOutputKeys[-1]
    model.fieldOutputRequests[key].setValues(variables=('U', 'V', 'A'))
    model.fieldOutputRequests[key].setValues(frequency=1)

# -------------- Add Gravitational Acceleration -------------
# Adds gravity to the entire model
def add_gravity(stepName):
    model = get_model()
    model.Gravity('Gravitational Acceleration', createStepName=stepName, distributionType=UNIFORM, comp2=-9.81)

# -------------- Add distributed mass -------------
# This function adds distributed mass with the specified magnitude to a set of faces in the specified part.
def add_mass_per_area(floorPart, set_of_faces, mass_per_area, mass_name):
    m = get_model()
    f = set_of_faces.faces
    r = regionToolset.Region(faces=f)
    floorPart.engineeringFeatures.NonstructuralMass(name=mass_name, region=r, units=MASS_PER_AREA, magnitude=mass_per_area)

# This function takes in a dict containing information about multiple masses and its corresponding levels, creates subsets of the floors for each mass and applies mass to the subsets.
def mass_from_dict(floorPart, floorSet, mass_dict, grid):
    for key in mass_dict.keys():
        name = key
        start_level, end_level, mass = mass_dict[key]
        s = set_of_selected_floors(floorPart, floorSet, start_level, end_level, grid)
```

```
        add_mass_per_area(floorPart, s, mass, key)

# -------------- Add concetrated mass -------------
# This function takes in a dict created containing information about non-structural point masses and applies them to the correct vertices of the grid.
def point_mass_from_dict(pointMassDict, grid):
    m = get_model()
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    for key in pointMassDict.keys():
        d = pointMassDict[key]
        part_string = d["part"]
        try:
            prt = m.parts[part_string]
            verts = prt.vertices
        except:
            print("Could not find part: "+part_string)
            continue
        magnitude = d["magnitude"]
        x_start, y_start, z_start = d["startPoint"]
        x_end, y_end, z_end = d["endPoint"]
        x_iter = range(x_start,x_end+1)
        y_iter = range(y_start,y_end+1)
        z_iter = range(z_start,z_end+1)
        vertices_lst = []
        for y_ind in y_iter:
            y = y_coord_lst[y_ind]
            for z_ind in z_iter:
                z = z_coord_lst[z_ind]
                for x_ind in x_iter:
                    x = x_coord_lst[x_ind]
                    vertices_lst += [verts.findAt((x,y,z),)]
        if len(vertices_lst) > 0:
            vertices_array = part.VertexArray(vertices_lst)
            reg = regionToolset.Region(vertices=vertices_array)
            prt.engineeringFeatures.PointMassInertia(name=key, region=reg, mass=magnitude)
            print('Point mass "'+key+'" applied to '+str(len(vertices_lst))+' vertice(s) on part: '+part_string)
        else:
            print('Could not create point mass "'+key+'" (no vertices were found).')
```

```python


# --------------  Import .txt table and create amplitude -------------
# This fuction takes in amplitude_str (a string containg the filename/path) of a
# .txt file containg a load amplitude (see ex. of such a file i the digital appendix)
# the amplitude in the file is imported to a amplitude-object in abaqus.
def amplitude_from_file(amplitude_str):
    f = open(amplitude_str, 'r')
    lines = f.readlines()
    seq = []
    for line in lines:
        line = line.split()
        line = line[0].split(',')
        pair = (float(line[0]), float(line[1]))
        seq.append(pair)
    f.close()
    model = get_model()
    model.TabularAmplitude(name='ImportedAmplitude', data=tuple(seq))
    return

# -------------- Create Job -------------
# This function creates a job and deletes and .lck files that might exist from previous analyses.
def create_and_run_job(jobName='TTBJob', run=True, nCpu=1, desc=''):
    if os.path.exists(jobName+'.lck'):
        os.remove(jobName+'.lck')

    mdb.Job(name=jobName, model=get_model(), numCpus=nCpu, numDomains=nCpu, description=desc)
    if run:
        mdb.jobs[jobName].submit(consistencyChecking=OFF)
        mdb.jobs[jobName].waitForCompletion()

# --------------  Create Pressure -------------
# This function creates a pressure load with the specified amplitude and magnitude
# and applies it to faces of the specified set in the specified part.
```

```
def create_pressure(load_name, shellPart, shellSet, mag, amplitude_name, step_name):
    model = get_model()
    a = get_assembly()
    shellInst = a.instances[shellPart.name]
    facesForLoad = shellSet.faces
    surfaceForLoad = shellPart.Surface(name=load_name+'_surface', side2Faces=facesForLoad)
    surf = shellInst.surfaces[load_name+'_surface']
    model.Pressure(amplitude=amplitude_name, createStepName=step_name, distributionType=UNIFORM, field='', magnitude=mag, name=load_name, region=surf)

# --------------  Create Free Vibration Initial Load -------------
# This function creates a impulse load to the upper edge of the top floor in the
# direction specified in xlsx_dict (imported from wind load sheet i excel file).
def free_vibration_impulse(xlsx_dict, floorPart, grid, stepName='FreeVibrationStep'):
    m = get_model()
    a = get_assembly()
    direction = xlsx_dict['WindDir']
    floorInst = a.instances[floorPart.name]
    e = floorPart.edges
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    if stepName in m.steps.keys():
        if direction.lower() == 'x':
            x = x_coord_matrix[0][-1]
            y = y_coord_lst[-1]
            z_min = z_coord_lst[0]
            z_max = z_coord_lst[-1]
            mag = 1000000/(z_max-z_min)
            edges_for_load = e.getByBoundingBox(xMin=x-0.01, yMin=y-0.01, zMin=z_min, xMax=x+0.01, yMax=y+0.01, zMax=z_max)
        elif direction.lower() == 'z':
            x_min = x_coord_matrix[-1][0]
            x_max = x_coord_matrix[-1][-1]
            y = y_coord_lst[-1]
            z = z_coord_lst[-1]
            mag = 1000000/(x_max-x_min)
```

```
            edges_for_load = e.getByBoundingBox(xMin=x_min,
             ↪  yMin=y-0.01, zMin=z-0.01, xMax=x_max, yMax=y+0.01,
             ↪  zMax=z+0.01)

        floorPart.Surface(name='FreeVibrationExcitation_surf',
         ↪  circumEdges=edges_for_load)
        surf = floorInst.surfaces['FreeVibrationExcitation_surf']
        amp_tuple = ((0,0), (0.2,1), (0.4,1), (0.41, 0))
        m.TabularAmplitude(name='FreeVibrationExcitation_amp',
         ↪  data=amp_tuple)
        m.ShellEdgeLoad(name='FreeVibrationExcitation',
         ↪  createStepName=stepName,
         ↪  amplitude='FreeVibrationExcitation_amp', magnitude=mag,
         ↪  region=surf)
```

C.4 TTB_boundaries.py

This file contains all the functions related to boundary conditions and interaction between the different parts of the structure.

```python
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys
import step

sys.path.append(scriptsFolder)

from TTB_general import *
from TTB_geometry import *

# -------------- Create Boundary Springs -------------
# This function creates springs (and dashpots) between ground and
↪ the frame structure
# REQUIRED ARGUMENTS:
# framePart - The part containing the frame
# bc_dict - A dictionary imported from excel containing the spring
↪ stiffnesses and dashpot coefs.
def create_boundary_springs_from_dict(framePart, bc_dict):
    verticesForSprings =
    ↪ framePart.vertices.getByBoundingBox(yMin=-0.01, yMax=0.01)
```

```python
    if verticesForSprings:
        regionForSprings = regionToolset.Region(vertices=verticesForSprings)
        for dof_nr in bc_dict.keys():
            spring_name, stiffness, dashpotCoef = bc_dict[dof_nr]
            if (stiffness > 0) and (dashpotCoef > 0):
                framePart.engineeringFeatures.SpringDashpotToGround(name=spring_name, region=regionForSprings, dof=dof_nr, springBehavior=ON, springStiffness=stiffness, dashpotBehavior=ON, dashpotCoefficient=dashpotCoef)
            elif stiffness > 0:
                framePart.engineeringFeatures.SpringDashpotToGround(name=spring_name, region=regionForSprings, dof=dof_nr, springBehavior=ON, springStiffness=stiffness)
            elif dashpotCoef > 0:
                framePart.engineeringFeatures.SpringDashpotToGround(name=spring_name, region=regionForSprings, dof=dof_nr, dashpotBehavior=ON, dashpotCoefficient=dashpotCoef)


# -------------- Create Ties Along Floor Edge -------------
# This function creates ties (connections) between the floors and the frame along the x-direction.
# REQUIRED ARGUMENTS:
# framePart - The part containing the frame
# floorPart - The part containing the floors
# grid -  List/matrix imported from excel containing coordinates of the axis system
# OPTIONAL ARGUMENTS:
# tie_rot - Boolean who specifies if rotations should be tied. Default is False.
def tie_floors(framePart, floorPart, grid, tie_rot=False, tol=0.01):
    model = get_model()
    a = get_assembly()
    frameInst = a.instances[framePart.name]
    floorInst = a.instances[floorPart.name]
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for y_ind in range(len(y_coord_lst)):
        y = y_coord_lst[y_ind]
        for z_ind in range(len(z_coord_lst)):
            z = z_coord_lst[z_ind]
```

```
            floorEdges = []
            beamEdges = []
            floorEdges = floorPart.edges.getByBoundingBox(yMin=y-tol, zMin=z-tol, yMax=y+tol, zMax=z+tol)
            if len(floorEdges)>0:
                beamEdges = framePart.edges.getByBoundingBox(yMin=y-tol, zMin=z-tol, yMax=y+tol, zMax=z+tol)
                if len(beamEdges)>0:
                    floorPart.Surface(name='floorSurf_y'+str(y_ind)+'_z'+str(z_ind), end1Edges=floorEdges)
                    framePart.Surface(name='frameSurf_y'+str(y_ind)+'_z'+str(z_ind), circumEdges=beamEdges)
                    floorSurf = floorInst.surfaces['floorSurf_y'+str(y_ind)+'_z'+str(z_ind)]
                    frameSurf = frameInst.surfaces['frameSurf_y'+str(y_ind)+'_z'+str(z_ind)]
                    tieName = floorPart.name+'_Tie_alongX_(YAXIS:'+str(y_ind)+'_ZAXIS:'+str(z_ind)+')'
                    if tie_rot:
                        model.Tie(name=tieName, master=floorSurf, slave=frameSurf, tieRotations=ON, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)
                    else:
                        model.Tie(name=tieName, master=floorSurf, slave=frameSurf, tieRotations=OFF, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)


#------------- Create Ties Along Floor Edge ---------
# This function creates ties (connections) between the floors and the frame along the x-direction.
def tie_floors_node_to_surf(floorPart, framePart, tie_rot=False):
    model = get_model()
    a  = get_assembly()
    frameInst = a.instances[framePart.name]
    floorInst = a.instances[floorPart.name]
    floorSurf = floorInst.surfaces['FloorSurfaces']
    frameEdges = frameInst.sets['XDirBeams']
```

```
    tieName = 'Floor_To_Frame_Tie'
    if tie_rot:
        model.Tie(name=tieName, master=floorSurf, slave=frameEdges, tieRotations=ON, adjust=OFF, constraintEnforcement=NODE_TO_SURFACE)
    else:
        model.Tie(name=tieName, master=floorSurf, slave=frameEdges, tieRotations=OFF, adjust=OFF, constraintEnforcement=NODE_TO_SURFACE)


# --------------  Create ties at corner of shells -------------
# This function creates ties (connections) between a shell part and the frame at the corners at each face in the shell part.
# The shells should be partitioned beforehand.
# REQUIRED ARGUMENTS:
# shell_part - The part containing the shell(s)
# frame_part - The part containing the frame
def create_corner_ties(shell_part, frame_part):
    model = get_model()
    a = get_assembly()
    frame_inst = a.instances[frame_part.name]
    shell_inst = a.instances[shell_part.name]
    shellFaces = shell_part.faces
    shellVertices = shell_part.vertices
    frameVertices = frame_part.vertices
    shellVertices_inst = shell_inst.vertices
    frameVertices_inst = frame_inst.vertices
    counter = 0
    tied_indices_list = []
    for f in shellFaces:
        v_list = f.getVertices()
        for v_ind in v_list:
            if v_ind not in tied_indices_list: # To aviod multiple ties at same vertice
                shellVertice = shellVertices[v_ind]
                coord = shellVertice.pointOn
                shellVertice = shellVertices_inst.findAt(coord)
                frameVertice = frameVertices_inst.findAt(coord, printWarning=False)
                if frameVertice:
                    frameRegion = regionToolset.Region(vertices=frameVertice)
```

```
                    shellRegion = regionToolset.Region(vertices=shellVertice)
                    tieName = 'CornerTie_'+shell_part.name+'_Vert_ind:'+str(v_ind)+'_(Counter:'+str(counter)+')'
                    model.Tie(master=frameRegion, slave=shellRegion, name=tieName, tieRotations=OFF, adjust=OFF)
                    counter += 1
                    tied_indices_list.append(v_ind)


# --------------  Create ties at the edges of shells -------------
# This function creates ties (connections) between a shell part and the frame at the edges of each face in the shell part.
# The shells should be partitioned beforehand.
# REQUIRED ARGUMENTS:
# shell_part - The part containing the shell(s)
# frame_part - The part containing the frame
def create_edge_ties(shell_part, frame_part, tie_rot=False):
    model = get_model()
    a = get_assembly()
    frame_inst = a.instances[frame_part.name]
    shell_inst = a.instances[shell_part.name]
    shellFaces = shell_part.faces
    shellVertices = shell_part.vertices
    shellEdges = shell_part.edges
    tied_shell_edge_ind = []
    for f in shellFaces:
        e_list = f.getEdges()
        for e_ind in e_list:
            if e_ind not in tied_shell_edge_ind:
                e = shellEdges[e_ind]
                shellEdges_Tie = part.EdgeArray([e])
                v_list = e.getVertices()
                coords = [shellVertices[v_ind].pointOn[0] for v_ind in v_list]
                frameEdges_Tie = frame_part.edges.getByBoundingCylinder(center1=coords[0], center2=coords[1], radius=0.1)
                if frameEdges_Tie:
                    tied_shell_edge_ind.append(e_ind)
                    frame_surf_name = frame_part.name+'_surf_(edge_'+str(e_ind)+')'
```

```
                    shell_surf_name = shell_part.name+'_surf_(edge_'+str(e_ind)+')'
                    frame_part.Surface(name=frame_surf_name, circumEdges=frameEdges_Tie)
                    shell_part.Surface(name=shell_surf_name, end1Edges=shellEdges_Tie)
                    frame_surf = frame_inst.surfaces[frame_surf_name]
                    shell_surf = shell_inst.surfaces[shell_surf_name]
                    tieName = frame_part.name+'_to_'+shell_part.name+'_Tie_(edge_'+str(e_ind)+')'
                    if tie_rot:
                        model.Tie(name=tieName, master=frame_surf, slave=shell_surf, tieRotations=ON, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)
                    else:
                        model.Tie(name=tieName, master=frame_surf, slave=shell_surf, tieRotations=OFF, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)


# --------------  Create ties at the edges of shells -------------
# This function creates ties (connections) between a shell part and the frame at the edges of each face in the shell part.
# The shells should be partitioned beforehand.
# REQUIRED ARGUMENTS:
# shell_part - The part containing the shell(s)
# frame_part - The part containing the frame
# OPTIONAL ARGUMENTS
# floor_part - If the user wish to create ties between the shell_part(i.e wall)
#              and the floors a floor_part may be specified. Ties between walls
#              and floors are only created if there is no beam at a wall panel edge.
# tie_rot - Boolean specifying if rotations should be restrained (if applicable)
def create_edge_ties2(shell_part, frame_part, floor_part=None, tie_rot=False):
    model = get_model()
    a = get_assembly()
```

```python
    frame_inst = a.instances[frame_part.name]
    shell_inst = a.instances[shell_part.name]
    shellFaces = shell_part.faces
    shellVertices = shell_part.vertices
    shellEdges = shell_part.edges
    try:
        floor_inst = a.instances[floor_part.name]
    except:
        pass
    tied_shell_edge_ind = []
    frame_counter = 0
    floor_counter = 0
    for f in shellFaces:
        e_list = f.getEdges()
        for e_ind in e_list:
            if e_ind not in tied_shell_edge_ind:
                e = shellEdges[e_ind]
                shellEdges_Tie_temp = part.EdgeArray([e])
                v_list = e.getVertices()
                coords = [shellVertices[v_ind].pointOn[0] for v_ind
                ↪ in v_list]
                frameEdges_Tie_temp = frame_part.edges.getByBoundin⌋
                ↪ gCylinder(center1=coords[0], center2=coords[1],
                ↪ radius=0.1)
                if frameEdges_Tie_temp:
                    tied_shell_edge_ind.append(e_ind)
                    if frame_counter == 0:
                        frameEdges_Tie = frameEdges_Tie_temp
                        shellEdges_frameTie = shellEdges_Tie_temp
                    else:
                        frameEdges_Tie += frameEdges_Tie_temp
                        shellEdges_frameTie += shellEdges_Tie_temp
                    frame_counter += 1
                elif floor_part:
                    mid_coord=tuple(np.divide(np.add(coords[0],
                    ↪ coords[1]), 2))
                    floorEdges_Tie_temp =
                    ↪ floor_part.edges.findAt(mid_coord,
                    ↪ printWarning=False)
                    if floorEdges_Tie_temp:
                        tied_shell_edge_ind.append(e_ind)
                        if floor_counter == 0:
                            floorEdges_Tie = [floorEdges_Tie_temp]
```

```
                            shellEdges_floorTie = shellEdges_Tie_temp
                        else:
                            floorEdges_Tie += [floorEdges_Tie_temp]
                            shellEdges_floorTie += shellEdges_Tie_temp
                        floor_counter += 1

    if frame_counter > 0:
        frame_surf_name = frame_part.name+'_surf_(tie_with_'+shell_part.name+')'
        shell_surf_name = shell_part.name+'_surf_(tie_with_'+frame_part.name+')'
        frame_part.Surface(name=frame_surf_name, circumEdges=frameEdges_Tie)
        shell_part.Surface(name=shell_surf_name, end1Edges=shellEdges_frameTie)
        frame_surf = frame_inst.surfaces[frame_surf_name]
        shell_surf = shell_inst.surfaces[shell_surf_name]
        tieName = frame_part.name+'_to_'+shell_part.name+'_Tie_(edge_'+str(e_ind)+')'
        if tie_rot:
            model.Tie(name=tieName, master=frame_surf, slave=shell_surf, tieRotations=ON, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)
        else:
            model.Tie(name=tieName, master=frame_surf, slave=shell_surf, tieRotations=OFF, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)

    if floor_counter > 0:
        floorEdges_Tie = part.EdgeArray(floorEdges_Tie)
        floor_surf_name = floor_part.name+'_surf_(tie_with_'+shell_part.name+')'
        shell_surf_name = shell_part.name+'_surf_(tie_with_'+floor_part.name+')'
        floor_part.Surface(name=floor_surf_name, end1Edges=floorEdges_Tie)
        shell_part.Surface(name=shell_surf_name, end1Edges=shellEdges_floorTie)
        floor_surf = floor_inst.surfaces[floor_surf_name]
        shell_surf = shell_inst.surfaces[shell_surf_name]
        tieName = floor_part.name+'_to_'+shell_part.name+'_Tie_(edge_'+str(e_ind)+')'
```

```
        if tie_rot:
            model.Tie(name=tieName, master=floor_surf, slave=shell_surf, tieRotations=ON, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)
        else:
            model.Tie(name=tieName, master=floor_surf, slave=shell_surf, tieRotations=OFF, adjust=OFF, constraintEnforcement=SURFACE_TO_SURFACE)

    print('Ties established between '+str(frame_counter)+' '+frame_part.name+' edges and '+shell_part.name+' edges.')
    print('Ties established between '+str(floor_counter)+' '+floor_part.name+' edges and '+shell_part.name+' edges.')


# --------------  Create connector panels (walls) -------------
# This function partitions each wall panel such that a "connection" zone is created at the edge of the panel. The connection zones can later be assigned material and sections that differs from the rest of the wall.
# This function are only functional for straight outer walls.
# REQUIRED ARGUMENTS:
# wallPart - The part containing the frame
# grid -  List/matrix imported from excel containing coordinates of the axis system
# width - The width of the connection zone to be created
def create_connector_panels_walls(shell_part, grid, width):
    model = get_model()
    x_coord_matrix, y_coord_lst, z_coord_lst = grid

    # XY-Plane
    for i in [0, len(z_coord_lst)-1]:
        z = z_coord_lst[i]
        x_coord_lst = x_coord_matrix[i]
        for j in range(1,len(x_coord_lst)):
            x_start = x_coord_lst[j-1]
            x_start_offset = x_start + width
            x_end = x_coord_lst[j]
            x_end_offset = x_end - width
            for k in range(1,len(y_coord_lst)):
                y_start = y_coord_lst[k-1]
                y_start_offset = y_start + width
                y_end = y_coord_lst[k]
                y_end_offset = y_end - width
```

```
                pt1 = (x_start_offset, y_start_offset)
                pt2 = (x_end_offset, y_end_offset)
                f = shell_part.faces.getByBoundingBox(xMin=x_start, yMin=y_start, zMin=z-0.001, xMax=x_end, yMax=y_end, zMax=z+0.001)
                shellPlane = create_principal_plane(XYPLANE, z, shell_part)
                shellUpEdge = create_principal_axis(YAXIS, shell_part)
                partitionTransform = shell_part.MakeSketchTransform(sketchPlane=shellPlane, origin=(0,0,z), sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge, sketchOrientation=LEFT)
                partitionSketch = model.ConstrainedSketch(name='partitionSketch', sheetSize=20, transform=partitionTransform)
                partitionSketch.rectangle(point1=(-pt1[0],pt1[1]), point2=(-pt2[0],pt2[1]))
                try:
                    shell_part.PartitionFaceBySketch(faces=f, sketchUpEdge=shellUpEdge, sketchOrientation=RIGHT, sketch=partitionSketch)
                except:
                    print('Could not partition wall panel in XY plane...')

    # YZ-Plane
    for j in [0,-1]:
        for i in range(1,len(z_coord_lst)):
            z_start = z_coord_lst[i-1]
            z_start_offset = z_start + width
            z_end = z_coord_lst[i]
            z_end_offset = z_end - width
            x_coord_lst = x_coord_matrix[i]
            x_start = x_coord_matrix[i-1][j]
            x_end = x_coord_matrix[i][j]
            x_min = min(x_start, x_end)
            x_max = max(x_start, x_end)
            for k in range(1,len(y_coord_lst)):
                y_start = y_coord_lst[k-1]
                y_start_offset = y_start + width
                y_end = y_coord_lst[k]
```

```python
                y_end_offset = y_end - width
                pt1 = (z_start_offset, y_start_offset)
                pt2 = (z_end_offset, y_end_offset)
                f = shell_part.faces.getByBoundingBox(xMin=x_min-0.001, yMin=y_start, zMin=z_start, xMax=x_max+0.001, yMax=y_end, zMax=z_end)
                shellPlane = create_principal_plane(YZPLANE, x_start, shell_part)
                shellUpEdge = create_principal_axis(YAXIS, shell_part)
                partitionTransform = shell_part.MakeSketchTransform(sketchPlane=shellPlane, origin=(x_start,0,0), sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge, sketchOrientation=RIGHT)
                partitionSketch = model.ConstrainedSketch(name='partitionSketch', sheetSize=20, transform=partitionTransform)
                partitionSketch.rectangle(point1=pt1, point2=pt2)
                try:
                    shell_part.PartitionFaceBySketch(faces=f, sketchUpEdge=shellUpEdge, sketchOrientation=RIGHT, sketch=partitionSketch)
                except:
                    print('Could not partition wall panel in YZ plane...')


# ------------------- Create Shaft Floor-Shaft Ties --------------------
# This function creates a tie between all shaft surfaces and floors.
# REQUIRED ARGUMENTS:
# shaftPart - Part object hosting the shafts
# floorPart - Part object hosting the floors
# Optional:
# tie_rot - Specifies if the rotational DOFs should be tied or not.
def shaft_floor_tie(shaftPart, floorPart, tie_rot = False):
    model = get_model()
    a = get_assembly()
    shaftPart.Surface(name='AllOuterShaftFaces', side2Faces=shaftPart.faces)
    floor_inst = a.instances[floorPart.name]
    shaft_inst = a.instances[shaftPart.name]
```

```
    tieName = 'Shaft_to_floor_tie'
    try:
        shaftSurf = shaft_inst.surfaces['AllOuterShaftFaces']
        floorEdges = floor_inst.sets['AllFloorEdgesAroundShafts']
        if tie_rot:
            model.Tie(name=tieName, master=floorEdges, slave=shaftSurf, adjust=OFF, tieRotations=ON, constraintEnforcement=NODE_TO_SURFACE, thickness=OFF)
        else:
            model.Tie(name=tieName, master=floorEdges, slave=shaftSurf, adjust=OFF, tieRotations=OFF, constraintEnforcement=NODE_TO_SURFACE, thickness=OFF)
    except KeyError:
        pass


# ------------------- Tie Walls ---------------
# Creates ties between the predefined surfaces "InnerSurface" and "EdgesForWallTies".
# Option to tie the rotational DOFs.
def wall_ties(wallPart, tie_rot=False):
    model= get_model()
    a = get_assembly()
    tieName = 'Wall Tie'
    wallInst = a.instances[wallPart.name]
    wallSurf = wallInst.surfaces['InnerSurface']
    tieEdges = a.sets['EdgesForWallTies']
    if tie_rot:
        model.Tie(name=tieName, master=wallSurf, slave=tieEdges, adjust=OFF, tieRotations=ON, constraintEnforcement=NODE_TO_SURFACE, thickness=OFF)
    else:
        model.Tie(name=tieName, master=wallSurf, slave=tieEdges, adjust=OFF, tieRotations=OFF, constraintEnforcement=NODE_TO_SURFACE, thickness=OFF)


# -------------- Tie the colomns and slab -------------
# Creates ties between the column ends and the slab.
def column_to_slab_tie(floorPart, framePart):
    model= get_model()
    a = get_assembly()
```

```
    tieName = 'Slab to Coloumns'
    floorInst = a.instances[floorPart.name]
    frameInst = a.instances[framePart.name]
    slabSurf = floorInst.surfaces['Slab Surface']
    columnNodes = frameInst.sets['Column Ends']

    model.Tie(name=tieName, master=slabSurf, slave=columnNodes,
     ↪  adjust=OFF, tieRotations=OFF,
     ↪  constraintEnforcement=NODE_TO_SURFACE, thickness=OFF)
```

C.5 TTB_excel.py

This file contains all the functions related to importing the input data from the input file generated in Excel. The data from Excel is mainly imported as dictionaries and lists to allow for further usage of the data inside python.

```
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys
import openpyxl
import unicodedata

sys.path.append(scriptsFolder)

from TTB_analysis import *

# --------------------- xz-grid coordinates --------
# This function creates a tuple containing a list of z-coordinates
    ↪ and a matrix of x-coordinates from a excel sheet.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the xz-grid
    ↪ data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
```

```python
# output_type - Specify the output type. 'coords' returns the
↪ coordinates, while 'lengths' returns the distances between the
↪ points in the grid.
# start_col - The first column containing user specified input.
↪ (Excel(1) indexing)
# start_row - The first row containing 1/0 (Excel(1) indexing)
def xz_grid_from_xlsx(sheet_name, wb_name, start_col=3,
↪ start_row=7):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]

    z_axis_vector = []
    z_iter = list(sheet.iter_cols(min_row=start_row-1,
     ↪ max_row=start_row-1, min_col=start_col, values_only=True))
    for z in z_iter:
        try:
            val = float(z[0])
            z_axis_vector.append(val)
        except:
            pass

    x_axis_vector = []
    x_iter = list(sheet.iter_rows(min_row=start_row,
     ↪ min_col=start_col-1, max_col=start_col-1, values_only=True))
    for x in x_iter:
        try:
            val = float(x[0])
            x_axis_vector.append(val)
        except:
            pass

    x_coord_matrix = []
    for i in range(len(z_axis_vector)):
        x_vec = []
        for j in range(len(x_axis_vector)):
            excel_i = start_col+i
            excel_j = start_row+j
            bol = sheet.cell(column=excel_i, row=excel_j).value
            if bol:
                x_coord = x_axis_vector[j]
                x_vec.append(x_coord)
        x_coord_matrix.append(x_vec)

    return (z_axis_vector, x_coord_matrix)
```

```python

# --------------------- y-grid coordinates --------
# This function creates a list of y-coordinates and a matrix.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the y-grid data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# output_type - Specify the output type. 'coords' returns the coordinates, while 'lengths' returns the distances between the points in the grid.
# start_col - The first column containing user specified input. (Excel(1) indexing)
# start_row - The first row containing user specified input. (Excel(1) indexing)
def y_grid_from_xlsx(sheet_name, wb_name, output_type='coords', start_col=2, start_row=5):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    y_coord_row = sheet[start_row]
    i = start_col - 1
    y_coord_vector = [float(y_coord_row[i].value)]
    i += 1
    while y_coord_row[i].value:
        val = y_coord_row[i].value
        y_coord_vector.append(float(val))
        i += 1

    if output_type.lower() == 'lengths':
        y_lengths_vector = []
        for i in range(1, len(y_coord_vector)):
            difference = y_coord_vector[i] - y_coord_vector[i - 1]
            y_lengths_vector.append(difference)
        return y_lengths_vector

    else:
        return y_coord_vector


# --------------------- (Beam-Type) Cross Sections --------
# This function creates a dictionary containg data about (beam-type) cross sections.
# REQUIRED ARGUMENTS:
```

```python
# sheet_name - The name of the excel sheet containing the cross section data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
def cross_section_dict_from_xlsx(sheet_name, wb_name, start_row=5):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    cs_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1, max_col=4, values_only=True):
        key_temp = row[0]
        if key_temp:
            key = unicodedata.normalize("NFKD", key_temp).encode("ascii", "ignore")
            dim_temp = row[1:3]
            dim = [float(x) for x in dim_temp]
            mat_temp = row[3]
            mat = unicodedata.normalize("NFKD", mat_temp).encode("ascii", "ignore")
            data = dim + [mat]
            cs_dict[key] = data
        else:
            continue
    return cs_dict


# --------------------- (Shell) Cross Sections --------
# This function creates a dictionary containg data about (shell-type) cross sections.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the cross section data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
def shell_section_dict_from_xlsx(sheet_name, wb_name, start_row=5):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    cs_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1, max_col=3, values_only=True):
```

```
        key_temp = row[0]
        if key_temp:
            key = unicodedata.normalize("NFKD", key_temp).encode("ascii", "ignore" )
            t = float(row[1])
            mat_temp = row[2]
            mat = unicodedata.normalize("NFKD", mat_temp).encode("ascii", "ignore" )
            cs_dict[key] = [t, mat]
        else:
            continue
    return cs_dict


# --------------------- Materials --------
# This function creates a dictionary containg data about all the materials specified.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the material data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
def create_material_dict_from_xlsx(sheet_name, wb_name, start_row=8):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    all_mat_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1, max_col=12, values_only=True):
        mat_dict = {}
        name_temp = row[0]
        if name_temp:
            name = unicodedata.normalize("NFKD", name_temp).encode("ascii", "ignore" )
            type_temp = row[1]
            type = unicodedata.normalize("NFKD", type_temp).encode("ascii", "ignore" )
            mat_dict['Type'] = type
            mat_dict['Density'] = float(row[2])
            mat_dict['E1'] = float(row[3])
            mat_dict['Nu12'] = float(row[6])
            if type in ['Trans. Isotropic', 'Orthotropic']:
```

```
                mat_dict['E2'] = float(row[4])
                mat_dict['Nu23'] = float(row[8])
                mat_dict['G12'] = float(row[9])
            if type == 'Orthotropic':
                mat_dict['E3'] = float(row[5])
                mat_dict['Nu_13'] = float(row[7])
                mat_dict['G13'] = float(row[10])
                mat_dict['G23'] = float(row[11])
            all_mat_dict[name] = mat_dict
        else:
            continue
    return all_mat_dict


# ---------------------   Elements --------
# This function creates a dictionary containg data about connector elements.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the connector data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing the name of the member. (Excel(1) indexing)
def create_connector_dict_from_xlsx2(sheet_name, wb_name, start_row=8):
        workbook = openpyxl.load_workbook(wb_name, data_only=True)
        sheet = workbook[sheet_name]
        connector_dict = {}
        for row in sheet.iter_rows(min_row=start_row, min_col=1, max_col=13, values_only=True):
            key_temp = row[0]
            if key_temp:
                key = unicodedata.normalize("NFKD", key_temp).encode("ascii", "ignore")
                segLen = float(row[1])
                fractions_temp = row[2:6]
                fractions = [float(x) for x in fractions_temp] # [Area, I11, I22, J]
                vals_temp = row[6:10]
                vals = [float(x) for x in vals_temp] # [Area, I11, I22, J]
                damping_temp = row[10:13]
                damping = [0]*len(damping_temp)
```

```python
                for i in range(len(damping_temp)):
                    try:
                        damping[i] = float(damping_temp[i])
                    except:
                        damping[i] = 0

                connector_dict[key] = [segLen, fractions, vals, damping]
            else:
                continue
        return connector_dict


# --------------------- Boundary Conditions --------
# This function creates a dictionary containg spring stiffnesses for ground springs.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the spring stiffness data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
def create_boundary_spring_dict_from_xlsx(sheet_name, wb_name, start_row=5):
        workbook = openpyxl.load_workbook(wb_name, data_only=True)
        sheet = workbook[sheet_name]
        bc_dict = {}
        for row in sheet.iter_rows(min_row=5, max_row=10, min_col=1, max_col=4, values_only=True):
            key = int(row[0])
            try:
                stiffness = float(row[2])
            except:
                stiffness = 0
            try:
                dashpotCoef = float(row[3])
            except:
                dashpotCoef = 0

            desc_temp = row[1]
            desc = unicodedata.normalize("NFKD", desc_temp).encode("ascii", "ignore")
            data = (desc, stiffness, dashpotCoef)
```

```
            bc_dict[key] = data
        return bc_dict


# --------------------- Diagonals --------
# This function creates a dictionary containg data about the placement of the diagonals.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the information about the diagonals.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
def diagonals_dict_from_xlsx(sheet_name, wb_name, start_row=4):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    all_diag_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1, max_col=9, values_only=True):
        diag_dict = {}
        name_temp = row[0]
        if name_temp:
            name = unicodedata.normalize("NFKD", name_temp).encode("ascii", "ignore" )
            plane_temp = row[1]
            plane = unicodedata.normalize("NFKD", plane_temp).encode("ascii", "ignore" )
            diag_dict['Plane'] = plane

            try:
                axis_lst = [int(row[2])]
            except:
                axis_string = unicodedata.normalize("NFKD", row[2]).encode("ascii", "ignore" )
                axis_lst = axis_string.split(';')
                axis_lst = [st.strip() for st in axis_lst]
                axis_lst = [int(st) for st in axis_lst]

            diag_dict['Axis'] = axis_lst
            diag_dict['Start Level'] = int(row[3])
            diag_dict['End Level'] = int(row[4])
            diag_dict['Start Column'] = int(row[5])
            diag_dict['End Column'] = int(row[6])
```

```
            try:
                diag_dict['Skip Levels'] = int(row[7])
            except:
                string = unicodedata.normalize("NFKD", row[7]).encode("ascii", "ignore" )
                lst = string.split(';')
                lst = [st.strip() for st in lst]
                lst = [int(st) for st in lst]
                diag_dict['Skip Levels'] = lst

            diag_dict['Intersect At'] = float(row[8])
            all_diag_dict[name] = diag_dict
        else:
            continue
    return all_diag_dict


#---------------------- Remove Coloumns/Beams ------------------
# This function creates a dictionary containing data about what beams and coloumns to remove from the original frame based on the grid.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the information about the diagonals.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
def remove_dict_from_xlsx(sheet_name, wb_name, start_row=4):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    all_remove_dict = {}
    name_ind = 1
    for row in sheet.iter_rows(min_row=start_row, min_col=1, max_col=10, values_only=True):
        remove_dict = {}
        part_temp = row[0]
        if part_temp:
            name = 'Remove '+str(name_ind)
            part =  unicodedata.normalize("NFKD", part_temp).encode("ascii", "ignore" )
            remove_dict['Parts'] = part
            plane_temp = row[1]
```

```
            plane = unicodedata.normalize("NFKD", plane_temp).encode("ascii", "ignore" )
            remove_dict['Plane'] = plane
            try:
                axis_lst = [int(row[2])]
            except:
                axis_string = unicodedata.normalize("NFKD", row[2]).encode("ascii", "ignore" )
                axis_lst = axis_string.split(';')
                axis_lst = [st.strip() for st in axis_lst]
                axis_lst = [int(st) for st in axis_lst]

            remove_dict['Axis'] = axis_lst
            remove_dict['Start Level'] = int(row[3])
            remove_dict['End Level'] = int(row[4])
            remove_dict['Start Column'] = int(row[5])
            remove_dict['End Column'] = int(row[6])
            if part == 'Columns' or part == 'Beams and Columns':
                remove_dict['Remove Start/End'] = int(row[7])

            all_remove_dict[name] = remove_dict
            name_ind += 1
        else:
            continue
    return all_remove_dict


# --------------------- Damping --------
# This function creates a dictionary containg damping parameters for materials.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the damping data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
def damping_dict_from_xlsx(sheet_name, wb_name, start_row=8):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    damping_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1, max_col=16, values_only=True):
        name_temp = row[0]
```

```
        if name_temp:
            name = unicodedata.normalize("NFKD",
             ↪  name_temp).encode("ascii", "ignore" )
            data = []
            for i in range(12,16):
                try:
                    data.append(float(row[i]))
                except:
                    data.append(0)
            damping_dict[name] = data
    return damping_dict


# --------------------- Non Structural Mass --------
# This function creates a dictionary containg the non structural
 ↪  mass data.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the mass data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input.
 ↪  (Excel(1) indexing)
def mass_dict_from_xlsx(sheet_name, wb_name, start_row=6):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    mass_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1,
     ↪  max_col=4, values_only=True):
        name_temp = row[0]
        if name_temp:
            name = unicodedata.normalize("NFKD",
             ↪  name_temp).encode("ascii", "ignore" )
            data = []
            for i in range(1,3):
                data.append(int(row[i]))
            data.append(float(row[3]))
            mass_dict[name] = data
    return mass_dict


# --------------------- Non Structural Point Mass --------
def point_mass_dict_from_xlsx(sheet_name, wb_name, start_row=6):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
```

```python
    mass_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1,
     ↪  max_col=9, values_only=True):
        name_temp = row[0]
        if name_temp:
            name = unicodedata.normalize("NFKD",
             ↪  name_temp).encode("ascii", "ignore" )
            mass_dict[name] = {}
            mass_dict[name]["part"] = unicodedata.normalize("NFKD",
             ↪  row[1]).encode("ascii", "ignore" )
            start_pt = []
            end_pt = []
            for i in range(2,5):
                start_pt.append(int(row[i]))
                end_pt.append(int(row[i+3]))
            mass_dict[name]["startPoint"] = start_pt
            mass_dict[name]["endPoint"] = end_pt
            mass_dict[name]["magnitude"] = float(row[8])
    return mass_dict


# --------------------- Floor Data --------
# This function creates a dictionary containg data about the floor
 ↪  cross sections.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the floor
 ↪  data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input.
 ↪  (Excel(1) indexing)
def floor_dict_from_xlsx(sheet_name, wb_name, start_row=5):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    floor_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1,
     ↪  max_col=9, values_only=True):
        name_temp = row[0]
        if name_temp:
            name = unicodedata.normalize("NFKD",
             ↪  name_temp).encode("ascii", "ignore")
            data = []
            for i in range(1,3):
                data.append(int(row[i])) # StartFloor - Endfloor
```

```
            data.append(float(row[3])) # Thickness
            mat_temp = row[4] # Material
            mat = unicodedata.normalize("NFKD",
             ↪  mat_temp).encode("ascii", "ignore")
            data.append(mat)
            data.append(int(row[5])) # Include Outer Beams
            data.append(int(row[6])) # Include connector segments
            if data[-1] == 1:
                data.append(float(row[7])) # Average width of floor
                 ↪  elements
            else:
                data.append(None)

            try:
                mat_dir = unicodedata.normalize("NFKD",
                 ↪  row[8]).encode("ascii", "ignore")
            except:
                mat_dir = None
            data.append(mat_dir)
            floor_dict[name] = data
    return floor_dict


# --------------------- Shafts --------
# This function creates a dictionary containg information about the
 ↪  shafts.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the shaft
 ↪  data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input.
 ↪  (Excel(1) indexing)
def shaft_dict_from_xlsx(sheet_name, wb_name, start_row=5):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    shaft_dict = {}
    for row in sheet.iter_rows(min_row=start_row, min_col=1,
     ↪  max_col=10, values_only=True):
        name_temp = row[0]
        if name_temp:
            name = unicodedata.normalize("NFKD",
             ↪  name_temp).encode("ascii", "ignore")
            sub_dict = {}
```

```
                sub_dict['Connect To Building'] = int(row[1])
                start_coord = []
                for i in range(2,4):
                    start_coord.append(float(row[i]))
                sub_dict['Start Coordinate'] = start_coord
                end_coord = []
                for i in range(4,6):
                    end_coord.append(float(row[i]))
                sub_dict['End Coordinate'] = end_coord
                sub_dict['Start Level'] = int(row[6])
                sub_dict['End Level'] = int(row[7])
                try:
                    sub_dict['End Level Offset'] = float(row[8])
                except:
                    sub_dict['End Level Offset'] = 0

                if row[9] == None:
                    sub_dict['Remove Wall'] = 0
                else:
                    sub_dict['Remove Wall'] = int(row[9])
                shaft_dict[name] = sub_dict
    return shaft_dict


# --------------------- Mesh --------
# This function creates a dictionary containg information about the mesh.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the mesh data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
# end_row - The final row containg mesh input.
def mesh_dict_from_xlsx(sheet_name, wb_name, start_row=5, end_row=8):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    mesh_dict = {}
    for row in sheet.iter_rows(min_row=start_row, max_row=end_row, min_col=1, max_col=3, values_only=True):
        name_temp = row[0]
        if name_temp:
```

```
            name = unicodedata.normalize("NFKD", name_temp).encode("ascii", "ignore")
            size = float(row[1])
            elType_temp = row[2]
            elType = unicodedata.normalize("NFKD", elType_temp).encode("ascii", "ignore")
            mesh_dict[name]=[size, elType]
    return mesh_dict


# -------------------- Steps --------
# This function creates analysis steps based on input in Excel file.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the step data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# start_row - The first row containing user specified input. (Excel(1) indexing)
# end_row - The final row containg mesh input.
def steps_from_xlsx(sheet_name, wb_name, start_row=12, end_row=16):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    prev_step_name = 'Initial'
    for row in sheet.iter_rows(min_row=start_row, max_row=end_row, values_only=True):
        type =  unicodedata.normalize("NFKD", row[0]).encode("ascii", "ignore")
        inc_bool = bool(row[1])
        try:
            step_desc = unicodedata.normalize("NFKD", row[6]).encode("ascii", "ignore")
        except:
            step_desc = ''
        if inc_bool:
            if type.lower() == 'static':
                create_static_step(name='StaticStep', prevStep=prev_step_name, desc=step_desc)
                prev_step_name = 'StaticStep'
            elif type.lower() == 'frequency':
                number_of_modes = int(row[2])
                try:
                    sim_bool = bool(row[5])
                except:
                    sim_bool = False
```

```
                create_freq_step(name='FrequencyStep', nModes=number_of_modes, prevStep=prev_step_name, desc=step_desc, SIMBased=sim_bool)
                prev_step_name = 'FrequencyStep'
            elif type.lower() == 'free vibration':
                time_step = float(row[3])
                dur = float(row[4])
                create_modal_dyn_step(name='FreeVibrationStep', prevStep=prev_step_name, desc=step_desc, period=dur, stepSize=time_step)
                prev_step_name = 'FreeVibrationStep'
            elif type.lower() == 'modal dynamics':
                time_step = float(row[3])
                dur = float(row[4])
                create_modal_dyn_step(name='ModalDynamicsStep', prevStep=prev_step_name, desc=step_desc, period=dur, stepSize=time_step)
                prev_step_name = 'ModalDynamicsStep'
            elif type.lower() == 'static (ec wind)':
                create_static_step(name='Static_Wind_Eurocode', prevStep=prev_step_name, desc=step_desc)
                prev_step_name = 'Static_Wind_Eurocode'
            else:
                print('Step type "' +type+ '" not defined in Python Script. Create it directly in Abaqus Cae or modify script...')


# --------------------- Job --------
# This function creates and runs Abaqus job based on input in Excel file.
# Returns a boolean (True if job is set to run automatically)
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the job data.
# wb_name - The path/name of the excel file.
# OPTIONAL ARGUMENTS:
# row_nr - The row containing user specified input. (Excel(1) indexing)
def job_from_xlsx(sheet_name, wb_name, row_nr=21):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    data_row = sheet[row_nr]
    data_row = [c.value for c in data_row]
```

```
    name_temp = data_row[0]
    name = unicodedata.normalize("NFKD", name_temp).encode("ascii", "ignore")
    create_bool = bool(data_row[1])
    if create_bool:
        run_bool = bool(data_row[2])
        cpu_int = int(data_row[3])
        desc_temp = data_row[4]
        try:
            description = unicodedata.normalize("NFKD", desc_temp).encode("ascii", "ignore")
        except:
            description = ''
        create_and_run_job(jobName=name, run=run_bool, nCpu=cpu_int, desc=description)
    return run_bool


# --------------------- Job from Excel except name--------
# This function creates and runs Abaqus job based on input in Excel file.
# REQUIRED ARGUMENTS:
# sheet_name - The name of the excel sheet containing the job data.
# wb_name - The path/name of the excel file.
# jobName - name of job.
# OPTIONAL ARGUMENTS:
# row_nr - The row containing user specified input. (Excel(1) indexing)
def job_from_xlsx_except_name(sheet_name, wb_name, jobName, row_nr=18):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    data_row = sheet[row_nr]
    data_row = [c.value for c in data_row]
    name_temp = data_row[0]
    name = jobName
    create_bool = bool(data_row[1])
    if create_bool:
        run_bool = bool(data_row[2])
        cpu_int = int(data_row[3])
        desc_temp = data_row[4]
        try:
            description = unicodedata.normalize("NFKD", desc_temp).encode("ascii", "ignore")
```

```
        except:
            description = ''
        create_and_run_job(jobName=name, run=run_bool, nCpu=cpu_int, desc=description)


# --------------------- Shell Connector dictionary from Excel --------
## This function creates a dictionary containing information on connector-zones
## shell type members
def create_shell_connector_dict_from_xlsx(sheet_name, wb_name, start_row=6):
        workbook = openpyxl.load_workbook(wb_name, data_only=True)
        sheet = workbook[sheet_name]
        connector_dict = {}
        for row in sheet.iter_rows(min_row=start_row, values_only=True):
            key_temp = row[0]
            sub_dict = {}
            if key_temp:
                key = unicodedata.normalize("NFKD", key_temp).encode("ascii", "ignore")
                section_temp = row[1:3]
                section = [float(x) for x in section_temp]
                material_temp = row[3]
                material = unicodedata.normalize("NFKD", material_temp).encode("ascii", "ignore")
                section.append(material)
                sub_dict['Section'] = section

                damping = []
                for i in range(4,8):
                    try:
                        damping.append(float(row[i]))
                    except:
                        damping.append(0)
                sub_dict['Damping'] = damping
                try:
                    sub_dict['ConnectTo'] = unicodedata.normalize("NFKD", row[8]).encode("ascii", "ignore")
                except:
                    sub_dict['ConnectTo'] = 'NA'
```

```python

                connector_dict[key] = sub_dict
            else:
                continue
        return connector_dict


# --------------------- Wind Parameters (Eurocode) --------
# This function creates a dictionary containing the input parameters in the Wind-sheet of the input file.
def ec_wind_param_from_xlsx(sheet_name, wb_name):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    wind_dict = {}

    wind_dict['WindDir'] = unicodedata.normalize("NFKD", sheet['B4'].value).encode("ascii", "ignore")

    wind_dict['LogDec_Struct'] = struct_param_wind(sheet[8])
    wind_dict['LogDec_Aero'] = struct_param_wind(sheet[9])
    wind_dict['NatFreq'] = struct_param_wind(sheet[10])
    wind_dict['ModeExponent'] = struct_param_wind(sheet[11])
    wind_dict['r'] = float(sheet['B12'].value)

    wind_dict['TerrainCat'] = int(sheet['B16'].value)

    wind_dict['v_b0'] = float(sheet['B20'].value)

    wind_dict['ReturnPeriod_Load'] = float(sheet['B24'].value)
    wind_dict['ReturnPeriod_Acc'] = float(sheet['B25'].value)


    for row in sheet.iter_rows(min_row=29, max_row=33, values_only=True):
        key = unicodedata.normalize("NFKD", row[0]).encode("ascii", "ignore")
        wind_dict[key] = float(row[1])

    wind_dict['SampleHeigth_Acc'] = float(sheet['B37'].value)
    return wind_dict

# This function reades the input of a structural parameter row of the wind-sheet.
def struct_param_wind(row):
```

```python
    if unicodedata.normalize("NFKD", row[2].value).encode("ascii", "ignore") == 'Abaqus Based':
        return 'Abaqus'
    elif unicodedata.normalize("NFKD", row[2].value).encode("ascii", "ignore") == 'Eurocode':
        return 'Eurocode'
    else:
        return float(row[1].value)



# ---------------------- Add To Frame ------------
# This function creates a dict containing the information provided in the "Add to frame" sheet of the excel file.
def add_to_frame_from_xlsx(sheet_name, wb_name, start_row=7, end_row=132):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    sheet = workbook[sheet_name]
    add_placement_dict = {}
    add_section_dict = {}
    add_orientation_dict = {}
    add_include_conn_dict = {}
    add_connector_dict = {}
    for row in sheet.iter_rows(min_row=start_row, max_row=end_row, min_col=1, max_col=28, values_only=True):
        key_temp = row[0]
        sub_placement_dict = {}
        sub_connector_dict = {}
        if key_temp:
            key = unicodedata.normalize("NFKD", key_temp).encode("ascii", "ignore")
            startPt = (float(row[1]), float(row[2]), float(row[3]))
            endPt = (float(row[4]), float(row[5]), float(row[6]))
            sub_placement_dict['Start Point'] = startPt
            sub_placement_dict['End Point'] = endPt

            width = float(row[7])
            height = float(row[8])
            material = unicodedata.normalize("NFKD", row[9]).encode("ascii", "ignore")
            section = [width, height, material]

            orient_str = unicodedata.normalize("NFKD", row[10]).encode("ascii", "ignore")
```

```
            orient_vect = orient_str.split(';')
            orient_vect = [comp.strip() for comp in orient_vect]
            orient_vect = (float(comp) for comp in orient_vect)
            orient_vect = tuple(orient_vect)

            include_conn = int(row[15])

            if include_conn:
                sub_connector_dict = {}
                segLen = float(row[16])
                fractions_temp = row[17:21]
                fractions = [float(x) for x in fractions_temp] # [Area, I11, I22, J]
                vals_temp = row[21:25]
                vals = [float(x) for x in vals_temp] # [Area, I11, I22, J]
                damping_temp = row[25:28]
                damping = [0]*len(damping_temp)
                for i in range(len(damping_temp)):
                    try:
                        damping[i] = float(damping_temp[i]) #[Alpha, Beta, Composite]
                    except:
                        damping[i] = 0
                sub_connector_dict = [segLen, fractions, vals, damping]
                add_connector_dict[key] = sub_connector_dict

            add_placement_dict[key] = sub_placement_dict
            add_section_dict[key] = section
            add_orientation_dict[key] = orient_vect
            add_include_conn_dict[key] = include_conn

        else:
            continue
    add_dicts = {'Placement': add_placement_dict, 'Section': add_section_dict,
                 'Orientation': add_orientation_dict, 'IncludeConn': add_include_conn_dict,
                 'Connector': add_connector_dict}
    return add_dicts


# ---------------------- Add To Frame ------------
```

```python
# This function creates a dict containing the information provided
↪ in the "floor-to-shaft conections" sheet of the excel file.
def floor_shaft_connection_from_xlsx(sheet_name, wb_name,
↪ start_row=5, end_row=14):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    floor_to_shaft_dict = {}
    sheet = workbook[sheet_name]
    for row in sheet.iter_rows(min_row=start_row, max_row=end_row,
    ↪ min_col=1, max_col=7, values_only=True):
        key_temp = row[0]
        if key_temp:
            sub_dict = {}
            key = unicodedata.normalize("NFKD",
            ↪ key_temp).encode("ascii", "ignore")
            section_temp = row[1:3]
            section = [float(x) for x in section_temp]
            material_temp = row[3]
            material = unicodedata.normalize("NFKD",
            ↪ material_temp).encode("ascii", "ignore")
            section.append(material)
            damping = []
            for i in range(4,8):
                try:
                    damping.append(float(row[i]))
                except:
                    damping.append(0)

            sub_dict['Section'] = section
            sub_dict['Damping'] = damping
            floor_to_shaft_dict[key] = sub_dict
    return floor_to_shaft_dict


# ---------------------- Add To Frame ------------
# This function adds the damping specified in the "Step Level
↪ Damping" sheet of excel to the respective steps.
def step_damping_from_xlsx(sheet_name, wb_name):
    workbook = openpyxl.load_workbook(wb_name, data_only=True)
    floor_to_shaft_dict = {}
    sheet = workbook[sheet_name]
    m = get_model()
    startRowFreeVib = 6
    endRowFreeVib = 17
    startRowModalDyn = 23
```

```
    endRowModalDyn = 34
    try:
        freeVibStep = m.steps['FreeVibrationStep']
        freeVibStepIsCreated = 1
    except:
        freeVibStepIsCreated = 0
    try:
        modalDynStep = m.steps['ModalDynamicsStep']
        modalDynStepIsCreated = 1
    except:
        modalDynStepIsCreated = 0

    ## Free Vibration Step - Direct Modal
    directDampingList = []
    for row in sheet.iter_rows(min_row=startRowFreeVib, max_row=endRowFreeVib, min_col=1, max_col=3, values_only=True):
        if row[0]:
            startMode, endMode = [int(x) for x in row[:2]]
            critDampingFactor = float(row[2])
            directDampingList.append((startMode,endMode,critDampingFactor))
    if len(directDampingList)>0 and freeVibStepIsCreated:
        directDampingTup = tuple(directDampingList)
        freeVibStep.setValues(directDamping=directDampingTup)

    ## Free Vibration Step - Composite Modal
    compositeDampingList = []
    for row in sheet.iter_rows(min_row=startRowFreeVib, max_row=endRowFreeVib, min_col=5, max_col=6, values_only=True):
        if row[0]:
            startMode, endMode = [int(x) for x in row[:2]]
            compositeDampingList.append((startMode,endMode))
    if len(compositeDampingList)>0 and freeVibStepIsCreated:
        compositeDampingTup = tuple(compositeDampingList)
        freeVibStep.setValues(compositeDamping=compositeDampingTup)

    ## Free Vibration Step - Rayleigh
    rayleighDampingList = []
    for row in sheet.iter_rows(min_row=startRowFreeVib, max_row=endRowFreeVib, min_col=8, max_col=11, values_only=True):
        if row[0]:
```

```python
            startMode, endMode = [int(x) for x in row[:2]]
            a, b = [float(x) for x in row[2:]]
            rayleighDampingList.append((startMode,endMode,a,b))
    if len(rayleighDampingList)>0 and freeVibStepIsCreated:
        rayleighDampingTup = tuple(rayleighDampingList)
        freeVibStep.setValues(rayleighDamping=rayleighDampingTup)

    ## Modal Dynamics Step - Direct Modal
    directDampingList = []
    for row in sheet.iter_rows(min_row=startRowModalDyn, max_row=endRowModalDyn, min_col=1, max_col=3, values_only=True):
        if row[0]:
            startMode, endMode = [int(x) for x in row[:2]]
            critDampingFactor = float(row[2])
            directDampingList.append((startMode,endMode,critDampingFactor))
    if len(directDampingList)>0 and modalDynStepIsCreated:
        directDampingTup = tuple(directDampingList)
        modalDynStep.setValues(directDamping=directDampingTup)

    ## Modal Dynamics Step - Composite Modal
    compositeDampingList = []
    for row in sheet.iter_rows(min_row=startRowModalDyn, max_row=endRowModalDyn, min_col=5, max_col=6, values_only=True):
        if row[0]:
            startMode, endMode = [int(x) for x in row[:2]]
            compositeDampingList.append((startMode,endMode))
    if len(compositeDampingList)>0 and modalDynStepIsCreated:
        compositeDampingTup = tuple(compositeDampingList)
        modalDynStep.setValues(compositeDamping=compositeDampingTup)

    ## Modal Dynamics Step - Rayleigh
    rayleighDampingList = []
    for row in sheet.iter_rows(min_row=startRowModalDyn, max_row=endRowModalDyn, min_col=8, max_col=11, values_only=True):
        if row[0]:
            startMode, endMode = [int(x) for x in row[:2]]
            a, b = [float(x) for x in row[2:]]
            rayleighDampingList.append((startMode,endMode,a,b))
    if len(rayleighDampingList)>0 and modalDynStepIsCreated:
        rayleighDampingTup = tuple(rayleighDampingList)
```

```
        modalDynStep.setValues(rayleighDamping=rayleighDampingTup)
```

C.6 TTB_general.py

This file contains basic functions for e.g. initializing the model and creating parts.

```
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys
import datetime
import step

sys.path.append(scriptsFolder)

from TTB_general import *

# -------------- Rename model -------------
## This function takes a new name as the input and renames the model.
## The new name of the model is also returned.
## Max one model in database is assumed.
def change_model_name(new_name):
    oldName = mdb.models.keys()[0]
    mdb.models.changeKey(fromName=oldName, toName=new_name)
    return new_name


```

```
# -------------- Return model -------------
## This function takes no input and returns the model.
## Max one model in database is assumed.
def get_model():
    modelKey = mdb.models.keys()[0]
    model = mdb.models[modelKey]
    return model


# -------------- Create and return part -------------
## This function creates and returns part.
## Input are the name of the part and dimensions (optional, default
↪ = 3D)
## Max one model in database is assumed.
def create_part(part_name,dimensions=3):
    model = get_model()
    if dimensions == 2:
        dim=TWO_D_PLANAR
    elif dimensions == 3:
        dim=THREE_D
    pt = model.Part(name=part_name, dimensionality=dim,
     ↪ type=DEFORMABLE_BODY)
    return pt


# -------------- Get Assembly -------------
# This function takes no input and returns the assembly.
# Max one model in database is assumed.
def get_assembly():
    model = get_model()
    assembly = model.rootAssembly
    return assembly


# -------------- Create and return instance -------------
# This function creates a instance from a part. The instance gets
↪ the same name as the part.
# REQUIRED ARGUMENTS:
# partToInstance - The part to be instanced.
# OPTIONAL ARGUMENTS:
# dependentMeshing - (ON/OFF) Controls if meshing should be
↪ dependent/independent
def create_instance(partToInstance, dependentMeshing=ON):
    a = get_assembly()
```

```
    instanceName = partToInstance.name
    inst = a.Instance(name=instanceName, part=partToInstance,
     ↪  dependent=dependentMeshing)
    return inst


# -------------- Regenerate assembly -------------
# This function updates the assembly/instances.
def assembly_regenerate():
    a = get_assembly()
    a.regenerate()


# ------------- Check if one value is close to equal to another
 ↪  ---------------
def isclose(a, b, rel_tol=1e-09, abs_tol=0.0):
    return abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)


# ------------- Close all open Odbs ---------------
def close_odbs():
    all_odb = session.odbs
    keysLst = all_odb.keys()
    for k in keysLst:
        odb = all_odb[k]
        odb.close()


# ------------- Get value from lst who is closest to K
 ↪  ---------------
def closest(lst, K):
    return lst[min(range(len(lst)), key = lambda i: abs(lst[i]-K))]


# ------------- Get current date and time ---------------
def get_date_and_time():
    d = datetime.datetime.now()
    timestr = d.strftime('%H:%M:%S')
    datestr = d.strftime('%d. %b %Y')
    return datestr+' '+timestr
```

C.7 TTB_geometry.py

This file contains all the functions related to generating the geometry of the building. Beams, columns, bracing, walls, floors etc. are created using the functions from this script.

```python
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys

sys.path.append(scriptsFolder)

from TTB_general import *

# -------------- Create planes -------------
# This function creates parallell planes to either the XY-, XZ-, and YZ-planes and places it with a userspecified offset from the placement of the pripinal plane.
# Input is what plane you want to create (XYPLANE, XZPLANE or YZPLANE), the planes offset value from origin, and what part_or_instance the datum plane is related to.
# Returns the created plane.
def create_principal_plane(principalPlane, offset, part_or_instance):
```

```
    part_or_instance.DatumPlaneByPrincipalPlane(principalPlane, offset)
    keysLst = part_or_instance.datums.keys()
    keysLst.sort()
    datumPlane = part_or_instance.datums[keysLst[-1]]
    return datumPlane


# -------------- Create principal axes -------------
## This function creates a datum axes of one of the principal axes (X,Y,Z).
## principalAxis input should be either XAXIS, YAXIS or ZAXIS.
## Returns the created axis.
def create_principal_axis(principalAxis,part_or_instance):
    datumAxis = part_or_instance.DatumAxisByPrincipalAxis(principalAxis)
    keysLst = part_or_instance.datums.keys()
    keysLst.sort()
    datumAxis = part_or_instance.datums[keysLst[-1]]
    return datumAxis

# -------------- Get coordinates -------------
## This functions takes the nth, mth an kth axis in X,Y,Z directions and lists containing the beam/col lengths in each direction.
## And returns a tuple with the coordinate of this point
def get_coordinates(xDiv, yDiv, zDiv, xLengths, yLengths, zLengths):
    xCoord = sum(xLengths[:xDiv])
    yCoord = sum(yLengths[:yDiv])
    zCoord = sum(zLengths[:zDiv])
    return (xCoord, yCoord, zCoord)

## This function takes the nth axis and member length in one direction and returns the position along that axis
def get_coordinate(nDiv, lengths):
    coord = sum(lengths[:nDiv])
    return coord

# -------------- Create Shell -------------
## Input plane as string ('xy'/'xz'/'yz'), points as tuple with 2 coordinates in the plane, the offset from the zero plane, the part to host the shells.
def create_shell(inPlane, pt1, pt2, planePosition, shellPart):
    model = get_model()
```

```python
    if inPlane.lower() == 'xy':
        shellPlane = create_principal_plane(XYPLANE, planePosition, shellPart)
        shellUpEdge = create_principal_axis(YAXIS,shellPart)
        shellTransform = shellPart.MakeSketchTransform(sketchPlane=shellPlane, origin=(0,0,planePosition))
        shellSketch = model.ConstrainedSketch(name='shellSketch', sheetSize=20, transform=shellTransform)
        shellSketch.rectangle(point1=pt1, point2=pt2)
        shellPart.Shell(sketch=shellSketch, sketchPlane=shellPlane, sketchPlaneSide=SIDE1, sketchUpEdge=shellUpEdge)

    elif inPlane.lower() == 'xz':
        shellPlane = create_principal_plane(XZPLANE, planePosition, shellPart)
        shellUpEdge = create_principal_axis(ZAXIS,shellPart)
        shellTransform = shellPart.MakeSketchTransform(sketchPlane=shellPlane, origin=(0,planePosition,0), sketchPlaneSide= SIDE2, sketchOrientation=LEFT, sketchUpEdge=shellUpEdge)
        shellSketch = model.ConstrainedSketch(name='shellSketch', sheetSize=20, transform=shellTransform)
        shellSketch.rectangle(point1=pt1, point2=pt2)

        shellPart.Shell(sketch=shellSketch, sketchPlane=shellPlane, sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge)

    elif inPlane.lower() == 'yz':
        shellPlane = create_principal_plane(YZPLANE, planePosition, shellPart)
        shellUpEdge = create_principal_axis(YAXIS,shellPart)
        shellTransform = shellPart.MakeSketchTransform(sketchPlane=shellPlane, origin=(planePosition,0,0), sketchPlaneSide = SIDE2, sketchOrientation=LEFT, sketchUpEdge=shellUpEdge)
        shellSketch = model.ConstrainedSketch(name='shellSketch', sheetSize=20, transform=shellTransform)
        shellSketch.rectangle(point1=pt1, point2=pt2)

        shellPart.Shell(sketch=shellSketch, sketchPlane=shellPlane, sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge, sketchOrientation = LEFT)

```

```python
    else:
        print('ERROR: Wrong plane setting, shell not created...')

# -------------- Create Floor -------------
# This function creates a floor at the specified level.
# REQUIRED ARGUMENTS:
# floorPart - The part set to host the floors
# grid -  List/matrix imported from excel containing coordinates of the axis system
# level - An integer specifing the level of the floor (0-indexed)
def create_floor(floorPart, grid, level):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    y_coord = y_coord_lst[level]
    for i in range(1,len(z_coord_lst)):
        x1 = x_coord_matrix[i-1][0]
        z1 = z_coord_lst[i-1]
        x2 = x_coord_matrix[i][-1]
        z2 = z_coord_lst[i]
        create_shell('xz', (x1,z1), (x2,z2), y_coord, floorPart)

# -------------- Create Walls -------------
# This function creates walls between two levels.
# REQUIRED ARGUMENTS:
# wallPart - The part set to host the walls
# grid -  List/matrix imported from excel containing coordinates of the axis system
# start_level - An integer specifing the lower level of the walls (0-indexed)
# end_level - An integer specifing the top level of the walls (0-indexed)
def create_walls(wallPart, start_level, end_level, grid):
    model = get_model()
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    y1 = y_coord_lst[start_level]
    y2 = y_coord_lst[end_level]
    plane = create_principal_plane(XZPLANE, y1, wallPart)
    upEdge = create_principal_axis(ZAXIS, wallPart)
    sketchTransform = wallPart.MakeSketchTransform(sketchPlane=plane, origin=(0,y1,0), sketchOrientation=LEFT, sketchPlaneSide=SIDE1, sketchUpEdge=upEdge)
    wallSketch = model.ConstrainedSketch(name='wallSketch', sheetSize=20, transform=sketchTransform)
    z_iter = list(range(1,len(z_coord_lst)))
```

```
    for i in z_iter:
        x1 = x_coord_matrix[i-1][0]
        z1 = z_coord_lst[i-1]
        x2 = x_coord_matrix[i][0]
        z2 = z_coord_lst[i]
        wallSketch.Line(point1=(-x1,z1), point2=(-x2,z2))

    x1 = x_coord_matrix[-1][0]
    z1 = z_coord_lst[-1]
    x2 = x_coord_matrix[-1][-1]
    z2 = z_coord_lst[-1]
    wallSketch.Line(point1=(-x1,z1), point2=(-x2,z2))

    z_iter.reverse()
    for i in z_iter:
        x1 = x_coord_matrix[i-1][-1]
        z1 = z_coord_lst[i-1]
        x2 = x_coord_matrix[i][-1]
        z2 = z_coord_lst[i]
        wallSketch.Line(point1=(-x1,z1), point2=(-x2,z2))

    x1 = x_coord_matrix[0][0]
    z1 = z_coord_lst[0]
    x2 = x_coord_matrix[0][-1]
    z2 = z_coord_lst[0]
    wallSketch.Line(point1=(-x1,z1), point2=(-x2,z2))

    wallPart.ShellExtrude(sketchPlane=plane, sketchPlaneSide=SIDE1, sketchUpEdge=upEdge, sketch=wallSketch, depth=y2-y1, sketchOrientation=LEFT)


# --------- Create shell by rectangular extrusion ----------
# This function create a shell from a rectangular extrosion.
# REQUIRED ARGUMENTS:
# drawingPlane - plane used to draw extrusion shape ('xy'/'xz'/'yz')
# pt1, pt2 - tuples of coordinates in drawingPlane defining rectangle.
# startPlaneCoord - start position of extrusion
# endPlaneCoord - end position of extrusion
def create_rectangular_shell_extrude(drawingPlane, pt1, pt2, startPlaneCoord, shellPart, endPlaneCoord):
    depth = abs(endPlaneCoord - startPlaneCoord)
    model = get_model()
```

```
    if drawingPlane.lower() == 'xy':
        plane = create_principal_plane(XYPLANE, startPlaneCoord, shellPart)
        upEdge = create_principal_axis(YAXIS, shellPart)
        sketchTransform = shellPart.MakeSketchTransform(sketchPlane=plane, origin=(0,0,startPlaneCoord))
        shellSketch = model.ConstrainedSketch(name='shellSketch', sheetSize=20, transform=sketchTransform)
        shellSketch.rectangle(point1=pt1, point2=pt2)
        shellPart.ShellExtrude(sketchPlane=plane, sketchPlaneSide=SIDE1, sketchUpEdge=upEdge, sketch=shellSketch, depth=depth, sketchOrientation=LEFT)
    if drawingPlane.lower() =='xz':
        plane = create_principal_plane(XZPLANE, startPlaneCoord, shellPart)
        upEdge = create_principal_axis(ZAXIS, shellPart)
        sketchTransform = shellPart.MakeSketchTransform(sketchPlane=plane, origin=(0,startPlaneCoord,0), sketchOrientation=LEFT, sketchPlaneSide=SIDE1, sketchUpEdge=upEdge)
        shellSketch = model.ConstrainedSketch(name='shellSketch', sheetSize=20, transform=sketchTransform)
        shellSketch.rectangle(point1=(-pt1[0],pt1[1]), point2=(-pt2[0],pt2[1]))
        shellPart.ShellExtrude(sketchPlane=plane, sketchPlaneSide=SIDE1, sketchUpEdge=upEdge, sketch=shellSketch, depth=depth, sketchOrientation=LEFT)
    if drawingPlane.lower() == 'yz':
        plane = create_principal_plane(YZPLANE, startPlaneCoord, shellPart)
        upEdge = create_principal_axis(YAXIS, shellPart)
        sketchTransform = shellPart.MakeSketchTransform(sketchPlane=plane, origin=(0,0,startPlaneCoord), sketchOrientation=LEFT, sketchPlaneSide=SIDE1, sketchUpEdge=upEdge)
        shellSketch = model.ConstrainedSketch(name='shellSketch', sheetSize=20, transform=sketchTransform)
        shellSketch.rectangle(point1=(-pt1[0],pt1[1]), point2=(-pt2[0],pt2[1]))
        shellPart.ShellExtrude(sketchPlane=plane, sketchPlaneSide=SIDE1, sketchUpEdge=upEdge, sketch=shellSketch, depth=depth, sketchOrientation=LEFT)
    del shellSketch
```

```python

# -------------- Partition Shells -------------
## This function partitions all faces in a given part by creating planes according to the grid.
## REQUIRED ARGUMENTS:
## shellPart - The part hosting the shells to be partitioned.
## grid - List of lists containg the grid system (x,y and z coordinates)
## cuttingPlaneOrientation - The orientation of the cutting plane. (XYPLANE, XZPLANE or YZPLANE)
def partition_shells(shellPart, grid, cuttingPlaneOrientation, selection_tol=0.1):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    if cuttingPlaneOrientation == XYPLANE:
        for i in range(1, len(z_coord_lst)-1):
            planePosition = z_coord_lst[i]
            dp = create_principal_plane(cuttingPlaneOrientation, planePosition, shellPart)
            f = shellPart.faces
            try:
                shellPart.PartitionFaceByDatumPlane(faces=f, datumPlane=dp)
            except:
                pass

    if cuttingPlaneOrientation == XZPLANE:
        for i in range(1, len(y_coord_lst)-1):
            planePosition = y_coord_lst[i]
            dp = create_principal_plane(cuttingPlaneOrientation, planePosition, shellPart)
            f = shellPart.faces
            try:
                shellPart.PartitionFaceByDatumPlane(faces=f, datumPlane=dp)
            except:
                pass

    if cuttingPlaneOrientation == YZPLANE:
        for i in range(len(z_coord_lst)):
            z = z_coord_lst[i]
            for j in range(1, len(x_coord_matrix[i])-1):
                planePosition = x_coord_matrix[i][j]
```

```
                dp = create_principal_plane(cuttingPlaneOrientation, planePosition, shellPart)
                f = shellPart.faces.getByBoundingBox(zMin=z-selection_tol, zMax=z+selection_tol)
                try:
                    shellPart.PartitionFaceByDatumPlane(faces=f, datumPlane=dp)
                except:
                    pass

## -------- Partition Shells at User Specified Plane --------
## This function partitions a shell part or a set of a shell part at a specified plane.
## REQUIRED ARGUMENTS:
## shellPart - the part hosting the shell to be partitioned
## planeOrientation - The orientation of the cutting plane. (XYPLANE, XZPLANE or YZPLANE)
## planePosition - The position of the cutting plane
## OPTIONAL ARGUMENTS:
## setName - name of the set the partition should be applied to.
##          If no input, partition is applied to entire part.
def partition_shells_specified(shellPart, planeOrientation, planePosition, setName = None):
    dp = create_principal_plane(planeOrientation, planePosition, shellPart)
    if setName == None:
        f = shellPart.faces
    else:
        f = shellPart.sets[setName].faces

    try:
        shellPart.PartitionFaceByDatumPlane(faces=f, datumPlane=dp)
    except:
        pass

# ----------- Create Connector Fields for Floors and Store Them as A Set--------
## This function creates partitions in the specified floors in order to simulate element connections.
## NOTE! There are only made partitions parallell to the span direction of the floor elements, The floor elements are modeled as continous in the span direction
```

```
## REQUIRED ARGUMENTS:
## floorPart - part osting the floors to be partitioned
## floor_dict - dicitonary containing information about floor sections
## grid - List of lists containg the grid system (x,y and z coordinates)
def floor_connector_partition(floorPart, floor_dict, shell_connector_dict, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    xWidth = abs(x_coord_lst[-1]-x_coord_lst[0])
    for key in floor_dict.keys():
        floor = floor_dict[key]
        if floor[5] == 1:
            connector = shell_connector_dict[key]
            section = connector['Section']
            connWidth = section[0]
            approxElemWidth = floor[6]
            numOfConn = int(xWidth/approxElemWidth)+1
            elemWidth = xWidth/(numOfConn+1)
            setName = key
            xCoord = elemWidth
            while xCoord < x_coord_lst[-1]-connWidth:
                for offset in [-connWidth/2, connWidth/2]:
                    cuttingPlanePosition = xCoord + offset
                    cuttingPlaneOrientation = YZPLANE
                    partition_shells_specified(floorPart, cuttingPlaneOrientation, cuttingPlanePosition, setName)
                xCoord += elemWidth

# ---------- Floor-to-shaft connector partition ------
## This function creates partitions of floors in order to specify properties at connection to shaft
## REQUIRED ARGUMENTS:
## floorPart - part hosting the floors to be partitioned
## shaft_dict - dictionary containing information about shafts
## floor_dict - dicitonary containing information about floor sections
# floor_to_shaft_dict - dicitonary containing information about floor to shaft connector
## grid - List of lists containg the grid system (x,y and z coordinates)
```

```
def floor_shaft_partition(floorPart, floor_dict, shaft_dict,
↪ floor_to_shaft_dict, grid):
    model = get_model()
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    tol = 0.01
    if floor_to_shaft_dict:
        for floor_key in floor_to_shaft_dict.keys():
            floor = floor_dict[floor_key]
            floor_shaft_conn = floor_to_shaft_dict[floor_key]
            startLevel_floor = floor[0]
            endLevel_floor = floor[1]
            connWidth = floor_shaft_conn['Section'][0]
            for shaft_key in shaft_dict.keys():
                shaft = shaft_dict[shaft_key]
                if shaft['Connect To Building']:
                    startLevel_shaft = shaft['Start Level']
                    endLevel_shaft = shaft['End Level']
                    if startLevel_shaft < startLevel_floor:
                        startLevel = startLevel_floor
                    else:
                        startLevel = startLevel_shaft
                    if endLevel_shaft < endLevel_floor:
                        endLevel = endLevel_shaft
                    else:
                        endLevel = endLevel_floor
                    yStart = y_coord_lst[startLevel]-tol
                    yEnd = y_coord_lst[endLevel]+tol
                    if startLevel_shaft == 0:
                        yStart = yStart+tol

                    xzStart_shaft = shaft['Start Coordinate']
                    xzEnd_shaft = shaft['End Coordinate']

                    xzStart_connector =
                    ↪ (xzStart_shaft[0]-connWidth,
                    ↪ xzStart_shaft[1]-connWidth)
                    xzEnd_connector = (xzEnd_shaft[0]+connWidth,
                    ↪ xzEnd_shaft[1]+connWidth)

                    cutFaces = floorPart.sets[floor_key].faces
                    shellPlane = create_principal_plane(XZPLANE,
                    ↪ yStart, floorPart)
```

```
                    shellUpEdge = create_principal_axis(ZAXIS,floorPart)
                    partitionTransform = floorPart.MakeSketchTransform(sketchPlane=shellPlane, origin=(0,yStart,0), sketchPlaneSide=SIDE2, sketchOrientation=LEFT, sketchUpEdge=shellUpEdge)
                    partitionSketch = model.ConstrainedSketch(name='partitionSketch', sheetSize=20, transform=partitionTransform)
                    partitionSketch.rectangle(point1=xzStart_connector,
                                              point2=xzEnd_connector)
                    floorPart.PartitionFaceBySketchDistance(faces=cutFaces, distance=yEnd-yStart, sketchPlane=shellPlane, sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge,sketchOrientation=LEFT, sketch=partitionSketch)
                    del partitionSketch
                else:
                    continue

# -------------- Create columns -------------
## This function creates columns in the XY-plane at a given Z-axis
## REQUIRED ARGUMENTS:
## colPart - The part to host the created beams and columns.
## grid - List of lists containg the grid system (x,y and z coordinates)
## z_axis_nr - The axis number of the created plane frame.
## segmentLength - The length of the created connector segments
def create_columns(colPart, grid, z_axis_nr):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    z = z_coord_lst[z_axis_nr]
    x_coord_lst = x_coord_matrix[z_axis_nr]
    colStartPts = []
    colEndPts = []
    for i in range(len(x_coord_lst)):
        x = x_coord_lst[i]
        for j in range(len(y_coord_lst)-1):
            yStart = y_coord_lst[j]
            yEnd = y_coord_lst[j+1]
            colStartPts.append((x, yStart, z))
```

```
            colEndPts.append((x, yEnd, z))
    listOfColPtsTuples = []
    for i in range(len(colStartPts)):
        listOfColPtsTuples.append((colStartPts[i],colEndPts[i]))
    tupleOfColPtsTuples = tuple(listOfColPtsTuples)
    colPart.WirePolyLine(points=tupleOfColPtsTuples)

# -------------- Create beams for system without diagonals --------------
# This functions creates beams in a specified plane, and is used for planes without diagonals.
# REQUIRED ARGUMENTS:
# beamPlane - string specifying the plane the beas will be placed in ('xy'/'yz')
# beamPart - specifying the part to host the beams
# grid - List of lists containg the grid system (x,y and z coordinates)
# axis_nr - the axis number specifying the placement of the beam plane
# segmentLength - length connector segments
# OPTIONAL ARGUMENTS:
# beamLevels - integer or list of integer with level numbers specifying at what levels
#              the beams should be placed drawn
#              Defulat input places beams at all levels, except level 0.
def create_beams(beamPlane, beamPart, grid, axis_nr, segmentLength, beamLevels='all'):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    beamPts = []
    beamStartPts = []
    beamEndPts = []

    if beamLevels == 'all':
        beamLevels = []
        for i in range(len(y_coord_lst)):
            if i == 0:
                continue
            beamLevels.append(i)
    if beamPlane.lower() == 'xy':
        z = z_coord_lst[axis_nr]
        for i in beamLevels:
            y = y_coord_lst[i]
```

```
            for j in range(len(x_coord_matrix[axis_nr])-1):
                xStart = x_coord_matrix[axis_nr][j]
                xStartSeg = xStart + segmentLength
                xEnd = x_coord_matrix[axis_nr][j+1]
                xEndSeg = xEnd - segmentLength
                beamPts = [(xStart, y, z), (xStartSeg, y, z), (xEndSeg, y, z), (xEnd, y, z)]
                for k in range(len(beamPts)-1):
                    beamStartPts.append(beamPts[k])
                    beamEndPts.append(beamPts[k+1])
    if beamPlane.lower() == 'yz':
        x = x_coord_lst[axis_nr]
        for i in beamLevels:
            if i == 0:
                continue
            y = y_coord_lst[i]
            for j in range(len(z_coord_lst)-1):
                zStart = z_coord_lst[j]
                zStartSeg = zStart + segmentLength
                zEnd = z_coord_lst[j+1]
                zEndSeg = zEnd - segmentLength
                beamPts = [(x, y, zStart), (x, y, zStartSeg), (x, y, zEndSeg), (x, y, zEnd)]
                for k in range(len(beamPts)-1):
                    beamStartPts.append(beamPts[k])
                    beamEndPts.append(beamPts[k+1])
    listOfBeamPtsTuples = []
    for i in range(len(beamStartPts)):
        listOfBeamPtsTuples.append((beamStartPts[i],beamEndPts[i]))
    tupleOfBeamPtsTuples = tuple(listOfBeamPtsTuples)
    beamPart.WirePolyLine(points=tupleOfBeamPtsTuples)

# -------------- Create beams with diagonal connector segments --------------
## This function create beams in YZ-plane. The beams are partitoned into segments near connections to columns and diagonals in order to modify the stiffness of these beam parts.
## REQUIRED ARGUMENTS:
## beamPlane - Specify the plane the beams should be span in ('xy'/'yz')
## beamPart - part to host the beams
## grid - list imported from excel file containing all coordinates used to draw beams and columns
```

```python
## axis_nr - number of the axis specifying the plane the beam will
↪  be placed in (numbering starting from 0)
## segmentLenght - the length of beam segments used to simulate
↪  Connections
## beamDiagIntersect - list containing the intersection points of
↪  the beams and diagonals in the plane. This is the output from
↪  function diagonal_intersections()
## OPTIONAL ARGUMENTS:
## beamLevels - list(or integer) specifying the levels at which the
↪  beams should be created. If no input, beam will be added to all
↪  levels
##              NOTE - even if a level is not listed, a beam will
↪  be created in the span where the diagonal intersects with the
↪  level.
def create_beams_diag(beamPlane, beamPart, grid, axis_nr,
↪  segmentLength, beamDiagIntersect, beamLevels='all'):
    beamStartPts = []
    beamEndPts = []
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)

    if beamLevels == 'all':
        beamLevels = []
        for i in range(len(y_coord_lst)):
            if i == 0:
                continue
            beamLevels.append(i)

    k = 0
    bl = 0
    status = 'proceed'
    if beamPlane.lower() == 'xy':
        z = z_coord_lst[axis_nr]
        for i in range(len(y_coord_lst)):
            y = y_coord_lst[i]
            for j in range(len(x_coord_matrix[axis_nr])-1):
                xStart = x_coord_matrix[axis_nr][j]
                xStartSeg = xStart + segmentLength
                xEnd = x_coord_matrix[axis_nr][j+1]
                xEndSeg = xEnd - segmentLength

                startPt = (xStart, y, z)
                startSegPt = (xStartSeg, y, z)
                endPt = (xEnd, y, z)
```

```
endSegPt = (xEndSeg, y, z)

if k < len(beamDiagIntersect):
    xIntersect = beamDiagIntersect[k][0]
    yIntersect = beamDiagIntersect[k][1]
    zIntersect = beamDiagIntersect[k][2]
    if z == zIntersect:
        if y == yIntersect:
            if xStart == xIntersect or xEnd ==
            ↪  xIntersect:
                k += 1
            elif xStart < xIntersect < xEnd:
                xLeftSeg = xIntersect -
                ↪  segmentLength
                xRightSeg = xIntersect +
                ↪  segmentLength

                leftSegPt = (xLeftSeg, y, z)
                intersectPt = (xIntersect, y, z)
                rightSegPt = (xRightSeg, y, z)

                if (xIntersect-xStart) <=
                ↪  2*segmentLength:
                    newSegmentLength =
                    ↪  (xIntersect-xStart)/2
                    xStartSeg = xStart +
                    ↪  newSegmentLength
                    startSegPt = (xStartSeg, y, z)

                    beamPts = [startPt, startSegPt,
                    ↪  intersectPt, rightSegPt,
                    ↪  endSegPt, endPt]
                elif (xEnd-xIntersect) <=
                ↪  2*segmentLength:
                    newSegmentLength =
                    ↪  (xEnd-xIntersect)/2
                    xEndSeg = xEnd -
                    ↪  newSegmentLength
                    endSegPt = (xEndSeg, y, z)

                    beamPts = [startPt, startSegPt,
                    ↪  leftSegPt, intersectPt,
                    ↪  endSegPt, endPt]
                else:
```

```
                                    beamPts = [startPt, startSegPt,
                                     ↪  leftSegPt, intersectPt,
                                     ↪  rightSegPt,endSegPt, endPt]
                                for l in range(len(beamPts)-1):
                                    if isclose(beamPts[l][0],
                                     ↪  beamPts[l+1][0], 1e-05):
                                        if isclose(beamPts[l][1],
                                         ↪  beamPts[l+1][1],
                                         ↪  1e-05):
                                            if isclose(beamPts[l][
                                             ↪  2],
                                             ↪  beamPts[l+1][2],
                                             ↪  1e-05):
                                                continue
                                    beamStartPts.append(beamPts[l])
                                    beamEndPts.append(beamPts[l+1])
                                k +=1
                                continue
                if bl < len(beamLevels):
                    if beamLevels[bl] == i:
                        beamPts = [startPt, startSegPt, endSegPt,
                         ↪  endPt]
                        for l in range(len(beamPts)-1):
                            beamStartPts.append(beamPts[l])
                            beamEndPts.append(beamPts[l+1])
                        status = 'proceed'
                    else:
                        status = 'wait'
            if status == 'proceed':
                bl += 1

    if beamPlane.lower() == 'yz':
        x = x_coord_lst[axis_nr]
        for i in range(len(y_coord_lst)):
            y = y_coord_lst[i]
            for j in range(len(z_coord_lst)-1):
                zStart = z_coord_lst[j]
                zStartSeg = zStart + segmentLength
                zEnd = z_coord_lst[j+1]
                zEndSeg = zEnd - segmentLength

                startPt = (x, y, zStart)
                startSegPt = (x, y, zStartSeg)
                endPt = (x, y, zEnd)
```

```
                endSegPt = (x, y, zEndSeg)

                if k < len(beamDiagIntersect):
                    xIntersect = beamDiagIntersect[k][0]
                    yIntersect = beamDiagIntersect[k][1]
                    zIntersect = beamDiagIntersect[k][2]
                    if x == xIntersect:
                        if y == yIntersect:
                            if zStart == zIntersect or zEnd == zIntersect:
                                k += 1
                            elif zStart < zIntersect < zEnd:
                                zLeftSeg = zIntersect - segmentLength
                                zRightSeg = zIntersect + segmentLength

                                leftSegPt = (x, y, zLeftSeg)
                                intersectPt = (x, y, zIntersect)
                                rightSegPt = (x, y, zRightSeg)

                                if (zIntersect-zStart) <= 2*segmentLength:
                                    newSegmentLength = (zIntersect-zStart)/2
                                    zStartSeg = zStart + newSegmentLength
                                    startSegPt = (x, y, zStartSeg)

                                    beamPts = [startPt, startSegPt, intersectPt, rightSegPt, endSegPt, endPt]
                                elif (zEnd-zIntersect) <= 2*segmentLength:
                                    newSegmentLength = (zEnd-zIntersect)/2
                                    zEndSeg = zEnd - newSegmentLength
                                    endSegPt = (x, y, zEndSeg)

                                    beamPts = [startPt, startSegPt, leftSegPt, intersectPt, endSegPt, endPt]
                                else:
```

```
                                    beamPts = [startPt, startSegPt, leftSegPt, intersectPt, rightSegPt,endSegPt, endPt]
                                for l in range(len(beamPts)-1):
                                    if isclose(beamPts[l][0], beamPts[l+1][0], 1e-05):
                                        if isclose(beamPts[l][1], beamPts[l+1][1], 1e-05):
                                            if isclose(beamPts[l][2], beamPts[l+1][2], 1e-05):
                                                continue
                                    beamStartPts.append(beamPts[l])
                                    beamEndPts.append(beamPts[l+1])
                                k +=1
                                continue
                if bl < len(beamLevels):
                    if beamLevels[bl] == i:
                        beamPts = [startPt, startSegPt, endSegPt, endPt]
                        for l in range(len(beamPts)-1):
                            beamStartPts.append(beamPts[l])
                            beamEndPts.append(beamPts[l+1])
                        status = 'proceed'
                    else:
                        status = 'wait'
            if status == 'proceed':
                bl += 1

    listOfBeamPtsTuples = []
    for i in range(len(beamStartPts)):
            listOfBeamPtsTuples.append((beamStartPts[i],beamEndPts[i]))
    tupleOfBeamPtsTuples = tuple(listOfBeamPtsTuples)
    beamPart.WirePolyLine(points=tupleOfBeamPtsTuples)


# -------------- Find Intersection Points of Diagonals -------------
# Find all points where diagonal intersects with beams and columns, and return them in two separate lists.
# These lists are used as input for drawing diagonals and beams.
# REQUIRED ARGUMENTS:
```

```python
# diagPlane - specify what plane the diagonals are placed in as a string ('xy'/'xz'/'yz')
# startAxis - specifying what axis the bottom of the diagonal starts
#              NOTE - choice of startAxis decide the direction of the diagonal
#                     The index of the startAxis is therefore not required to
#                     be less than the index of the endAxis.
# endAxis - specifying end axis as boundary for diagonal.
# startLevel - lowest level of diagonal
# endLevel - top level of diagonal
# grid - list imported from excel file containing all coordinates used to draw beams and columns
# skipLevels - list or integer specifying the number of levels each diagonal span across
# diagAxis - integer specifying the axis of the diagonal plane.
# diagPart - specify the part to host the diagonal
# intersectAt - specify the position of the diagonal ends. Should be in the range 0-1, where 0 indicates that the diagonals ends in the point where the beam intersects the column (no offset),and 1 indicates that diagonals end at the level below (max offset).
def diagonal_intersections(diagPlane, startAxis, endAxis, startLevel, endLevel,
                           skipLevels, grid, diagAxis, diagPart, intersectAt):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    beamDiagIntersect = []
    colDiagIntersect = []

    # Check input
    if (type(skipLevels) is list) or (type(skipLevels) is tuple):
        if sum(skipLevels) != (endLevel-startLevel):
            print('ERROR: The sum of skipLevels is not equal to endLevel-startLevel!')
    elif type(skipLevels) is int:
        skipLevelsFloat = float(skipLevels)
        numOfDiags = np.ceil((endLevel-startLevel)/skipLevelsFloat)
        skipLevels = [skipLevels]*numOfDiags
    else:
        print('ERROR: skipLevels is neither a list or integer!')
    skipSpans = abs(endAxis - startAxis)
    if diagPlane.lower() == 'xy':
```

```
        xCoords = x_coord_matrix[diagAxis]
        yCoords = y_coord_lst
    if diagPlane.lower() == 'xz':
        xCoords = x_coord_matrix[diagAxis]
        yCoords = z_coord_lst
    if diagPlane.lower() == 'yz':
        xCoords = z_coord_lst
        yCoords = y_coord_lst
    xStart = xCoords[startAxis]
    xEnd = xCoords[endAxis]
    j = startLevel
    i = 0

    while j < endLevel:
        yStart = (1-intersectAt)*yCoords[j]+intersectAt*yCoords[j-1]
        if  j+skipLevels[i] < len(yCoords)-1:
            yEnd = (1-intersectAt)*yCoords[j+skipLevels[i]]+intersectAt*yCoords[j-1+skipLevels[i]]
        else:
            yEnd = yCoords[endLevel]
        if j == startLevel:
            yStart = 0
        diagIncl = abs((yEnd-yStart)/(xEnd-xStart))
        # Create list of beam-diagonal intersection points, except points where diagonals intersect with both columns and beams
        for k in range(skipLevels[i]+1):
            # Find intersection points
            if (j+k) > len(yCoords)-1:
                continue
            yBeamIntersect = yCoords[j+k]
            if xEnd > xStart:
                xBeamIntersect = xStart + (yBeamIntersect-yStart)/diagIncl
                if xBeamIntersect > xEnd:
                    continue
            else:
                xBeamIntersect = xStart - (yBeamIntersect-yStart)/diagIncl
                if xBeamIntersect < xEnd:
                    continue
```

```
            # Avoid saving points where diagonal intersect beam and coloumn at same place, and start/end point of diagonal
            if (k == 0 or k == skipLevels[i]) and intersectAt == 0:
                continue
            if yBeamIntersect == yCoords[startLevel] or yBeamIntersect == yCoords[endLevel]:
                continue
            if diagPlane.lower() == 'xy':
                if xBeamIntersect in x_coord_lst:
                    continue
                beamDiagIntersect.append((xBeamIntersect, yBeamIntersect, z_coord_lst[diagAxis]))
            if diagPlane.lower() == 'xz':
                if xBeamIntersect in x_coord_lst:
                    continue
                beamDiagIntersect.append((xBeamIntersect, y_coord_lst[diagAxis], yBeamIntersect))
            if diagPlane.lower() == 'yz':
                if xBeamIntersect in z_coord_lst:
                    continue
                beamDiagIntersect.append((x_coord_lst[diagAxis], yBeamIntersect, xBeamIntersect))
        # Create list of all coloumn-diagonal intersection points
        for k in range(skipSpans+1):
            # Find intersection points
            if xEnd > xStart:
                index = min(startAxis, endAxis)+k
                xColIntersect = xCoords[index]
            elif xEnd < xStart:
                index = max(startAxis, endAxis)-k
                xColIntersect = xCoords[index]
            yColIntersect = yStart + diagIncl*abs(xColIntersect-xStart)

            colCoords = coloumn_coords(grid)

            # Save points to list
            if diagPlane.lower() == 'xy':
                if (xColIntersect, yColIntersect, z_coord_lst[diagAxis]) not in colDiagIntersect:
                    colDiagIntersect.append((xColIntersect,yColIntersect, z_coord_lst[diagAxis]))
```

```
            if diagPlane.lower() == 'xz':
                if (xColIntersect, y_coord_lst[diagAxis], yColIntersect) not in colDiagIntersect:
                    colDiagIntersect.append((xColIntersect, y_coord_lst[diagAxis],yColIntersect))
                elif xColIntersect == xStart or xColIntersect == xEnd:
                    print('ERROR: Diagonal must be connected to coloumn at turning point')
            if diagPlane.lower() == 'yz':
                if (x_coord_lst[diagAxis],z_coord_lst[index]) in colCoords:
                    if (x_coord_lst[diagAxis], yColIntersect, xColIntersect) not in colDiagIntersect:
                        colDiagIntersect.append((x_coord_lst[diagAxis],yColIntersect,xColIntersect))
                elif xColIntersect == xStart or xColIntersect == xEnd:
                    print('ERROR: Diagonal must be connected to coloumn at turning point')

        xStart_temp = xStart
        xStart = xEnd
        xEnd = xStart_temp
        j += skipLevels[i]
        i += 1
    return beamDiagIntersect, colDiagIntersect

# --------------  Draw Diagonals -------------
## This function creates a diagonal in a sepcified plane.
## The diagonals will have separated segments close to intersection with columns in order to simulate connection behaviour
## REQUIRED ARGUMENTS:
## diagPlane - specifying the plane of the diagonal ('xy'/'xz'/'yz')
## diagPart - specify the part to host the diagonal
## colDiagIntersect - list containg all points of intersections between diagonal and columns
##                    The list is one of the outputs of the function diagonal_intersections()
## segmentLength - specify length of the connection segments

def draw_diagonals(diagPlane, diagPart, colDiagIntersect, segmentLength):
    listOfDiagPoints = []
```

```
    for i in range(len(colDiagIntersect)-1):
        startPt = colDiagIntersect[i]
        endPt = colDiagIntersect[i+1]

        if diagPlane.lower() == 'xy':
            diagIncl = abs((endPt[1]-startPt[1])/(endPt[0]-startPt[0]))
            InclAngle = np.arctan(diagIncl)
            if InclAngle == 0:
                continue
            if startPt[0] < endPt[0]:
                xStartSegPt = startPt[0] + np.cos(InclAngle)*segmentLength
                xEndSegPt = endPt[0] - np.cos(InclAngle)*segmentLength
            elif startPt[0] > endPt[0]:
                xStartSegPt = startPt[0] - np.cos(InclAngle)*segmentLength
                xEndSegPt = endPt[0] + np.cos(InclAngle)*segmentLength

            yStartSegPt = startPt[1] + np.sin(InclAngle)*segmentLength
            yEndSegPt = endPt[1] - np.sin(InclAngle)*segmentLength

            startSegPt = (xStartSegPt, yStartSegPt, startPt[2])
            endSegPt = (xEndSegPt, yEndSegPt, startPt[2])
        elif diagPlane.lower() == 'xz':
            diagIncl = abs((endPt[2]-startPt[2])/(endPt[0]-startPt[0]))
            InclAngle = np.arctan(diagIncl)
            if InclAngle == 0:
                continue
            if startPt[0] < endPt[0]:
                xStartSegPt = startPt[0] + np.cos(InclAngle)*segmentLength
                xEndSegPt = endPt[0] - np.cos(InclAngle)*segmentLength
            elif startPt[0] > endPt[0]:
                xStartSegPt = startPt[0] - np.cos(InclAngle)*segmentLength
                xEndSegPt = endPt[0] + np.cos(InclAngle)*segmentLength

```

```
            zStartSegPt = startPt[2] + np.sin(InclAngle)*segmentLength
            zEndSegPt = endPt[2] - np.sin(InclAngle)*segmentLength

            startSegPt = (xStartSegPt, startPt[1], zStartSegPt)
            endSegPt = (xEndSegPt, startPt[1], zEndSegPt)
        elif diagPlane.lower() == 'yz':
            diagIncl = abs((endPt[1]-startPt[1])/(endPt[2]-startPt[2]))
            InclAngle = np.arctan(diagIncl)
            if InclAngle == 0:
                continue
            if startPt[2] < endPt[2]:
                zStartSegPt = startPt[2] + np.cos(InclAngle)*segmentLength
                zEndSegPt = endPt[2] - np.cos(InclAngle)*segmentLength
            elif startPt[2] > endPt[2]:
                zStartSegPt = startPt[2] - np.cos(InclAngle)*segmentLength
                zEndSegPt = endPt[2] + np.cos(InclAngle)*segmentLength

            yStartSegPt = startPt[1] + np.sin(InclAngle)*segmentLength
            yEndSegPt = endPt[1] - np.sin(InclAngle)*segmentLength

            startSegPt = (startPt[0], yStartSegPt, zStartSegPt)
            endSegPt = (startPt[0], yEndSegPt, zEndSegPt)

        listOfDiagPoints.append((startPt, startSegPt))
        listOfDiagPoints.append((startSegPt, endSegPt))
        listOfDiagPoints.append((endSegPt, endPt))
    diagPart.WirePolyLine(points=listOfDiagPoints)

# ------------------ Create Shaft ---------------
# This function creates an elevator shaft
# NOTE - The function must be exectued AFTER sets have been created
# REQUIRED ARGUMENTS:
# shaftPart - part to host the shaft
# floorPart - part containing the floors to which holes will be added
# framePart - part containing beams that need to be removed in order to make room for shaft
```

```python
# shaf_dict - dictionary containg all relevant information regarding shaft geometry (generated from input file)
# grid - list imported from excel file containing all coordinates used to draw beams and columns
def create_shafts(shaftPart, floorPart, framePart, shaft_dict, grid, tol=0.01):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for key in shaft_dict.keys():
        shaft = shaft_dict[key]
        pt1 = tuple(shaft['Start Coordinate'])
        pt2 = tuple(shaft['End Coordinate'])
        startLevel = shaft['Start Level']
        endLevel = shaft['End Level']
        endLevelOffset = shaft['End Level Offset']
        removeWall = shaft['Remove Wall']
        yStart = y_coord_lst[startLevel]
        yEnd = y_coord_lst[endLevel]+endLevelOffset
        if shaft['Connect To Building']:
            create_rectangular_shell_extrude('xz', pt1, pt2, yStart, shaftPart, yEnd)
            if removeWall > 0:
                if removeWall == 1:
                    face = shaftPart.faces.getByBoundingBox(xMin=pt1[0]-tol, zMin=pt1[1]-tol, xMax=pt2[0]+tol, zMax=pt1[1]+tol)
                if removeWall == 2:
                    face = shaftPart.faces.getByBoundingBox(xMin=pt2[0]-tol, zMin=pt1[1]-tol, xMax=pt2[0]+tol, zMax=pt2[1]+tol)
                if removeWall == 3:
                    face = shaftPart.faces.getByBoundingBox(xMin=pt1[0]-tol, zMin=pt2[1]-tol, xMax=pt2[0]+tol, zMax=pt2[1]+tol)
                if removeWall == 4:
                    face = shaftPart.faces.getByBoundingBox(xMin=pt1[0]-tol, zMin=pt1[1]-tol, xMax=pt1[0]+tol, zMax=pt2[1]+tol)
                shaftPart.RemoveFaces(deleteCells=False, faceList=face)
        if startLevel == 0:
            yStart_cut = yStart+tol
        else:
            yStart_cut = yStart
```

```
        rectangular_cutout('xz', pt1, pt2, floorPart, yStart_cut, yEnd)
        pt1 = (pt1[0], yStart, pt1[1])
        pt2 = (pt2[0], yEnd, pt2[1])
        remove_edges_within_box(framePart, pt1, pt2, setName='InnerBeams')


# ------------------ Create Rectangular Cutout of Face --------------
# This function create a rectangular hole in a number of faces in the same part.
# REQUIRED ARGUMENTS:
# inPlane - specify what plane the cut should be made in ('xy'/'xz'/'yz')
# pt1, pt2 - points defining rectangle
# shellPart - specify what part the cut should be applied to
# startPlanePos - specify position of the start plane of the cut
# endPlanePos - specify the position of the end plane of the cut
def rectangular_cutout(inPlane, pt1, pt2, shellPart, startPlanePos, endPlanePos):
    model = get_model()
    cutFaces = shellPart.faces

    if inPlane.lower() == 'xy':
        shellPlane = create_principal_plane(XYPLANE, startPlanePos, shellPart)
        shellUpEdge = create_principal_axis(YAXIS,shellPart)
        partitionTransform = shellPart.MakeSketchTransform(sketchPlane=shellPlane, origin=(0,0,startPlanePos), sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge, sketchOrientation=RIGHT)
        partitionSketch = model.ConstrainedSketch(name='partitionSketch', sheetSize=20, transform=partitionTransform)
        partitionSketch.rectangle(point1=(-pt1[0],pt1[1]), point2=(-pt2[0],pt2[1]))
        shellPart.PartitionFaceBySketchDistance(faces=cutFaces, distance=endPlanePos-startPlanePos, sketchPlane=shellPlane, sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge, sketchOrientation=RIGHT, sketch=partitionSketch)
```

```
        partitionFaces = shellPart.faces.getByBoundingBox(xMin=pt1[0], yMin=pt1[1], zMin=0, xMax=pt2[0], yMax=pt2[1], zMax=endPlanePos)

    elif inPlane.lower() == 'xz':
        shellPlane = create_principal_plane(XZPLANE, startPlanePos, shellPart)
        shellUpEdge = create_principal_axis(ZAXIS,shellPart)
        partitionTransform = shellPart.MakeSketchTransform(sketchPlane=shellPlane, origin=(0,startPlanePos,0), sketchPlaneSide= SIDE2,sketchOrientation=LEFT,sketchUpEdge=shellUpEdge)
        partitionSketch = model.ConstrainedSketch(name='partitionSketch', sheetSize=20,transform=partitionTransform)
        partitionSketch.rectangle(point1=pt1, point2=pt2)
        shellPart.PartitionFaceBySketchDistance(faces=cutFaces, distance=endPlanePos-startPlanePos, sketchPlane=shellPlane, sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge, sketchOrientation=LEFT, sketch=partitionSketch)
        partitionFaces = shellPart.faces.getByBoundingBox(xMin=pt1[0], yMin=0, zMin=pt1[1], xMax=pt2[0], yMax=endPlanePos, zMax=pt2[1])

    elif inPlane.lower() == 'yz':
        shellPlane = create_principal_plane(YZPLANE, startPlanePos, shellPart)
        shellUpEdge = create_principal_axis(YAXIS,shellPart)
        partitionTransform = shellPart.MakeSketchTransform(sketchPlane=shellPlane, origin=(startPlanePos,0,0), sketchPlaneSide = SIDE2, sketchOrientation=LEFT, sketchUpEdge=shellUpEdge)
        partitionSketch = model.ConstrainedSketch(name='partitionSketch', sheetSize=20, transform=partitionTransform)
        partitionSketch.rectangle(point1=pt1, point2=pt2)
        shellPart.PartitionFaceBySketchDistance(faces=cutFaces, distance=endPlanePos-startPlanePos, sketchPlane=shellPlane, sketchPlaneSide=SIDE2, sketchUpEdge=shellUpEdge, sketchOrientation=LEFT, sketch=partitionSketch)
```

```
        partitionFaces = shellPart.faces.getByBoundingBox(xMin=0, yMin=pt1[1], zMin=pt1[0], xMax=endPlanePos, yMax=pt2[1], zMax=pt2[0])
    else:
        print('ERROR: Wrong plane setting, cutout not created...')
    del partitionSketch
    shellPart.RemoveFaces(faceList=partitionFaces)

# ---------------- Remove Edges Within Bounding Box --------------
# This function removes the edges of a specified set of a part, that lie within a user specified bounding box.
# REQUIRED ARGUMENTS:
# framePart - the part that the changes should be applied to
# pt1 - lower bound of bounding box
# pt2 - upper bound of bounding box
# OPTIONAL ARGUMENT:
# setName - name of set of which edges will be removed.(NOTE: Must be a string)
#           "None" input causes the command to be applied to entire part.

def remove_edges_within_box(framePart, pt1, pt2, setName=None):
if setName == None:
        wireEdges = framePart.edges.getByBoundingBox(xMin=pt1[0]-0.001, yMin=pt1[1]-0.001, zMin=pt1[2]-0.001, xMax=pt2[0]+0.001, yMax=pt2[1]+0.001, zMax=pt2[2]+0.001)
    else:
        wireEdges = framePart.sets[setName].edges.getByBoundingBox(xMin=pt1[0]-0.001, yMin=pt1[1]-0.001, zMin=pt1[2]-0.001, xMax=pt2[0]+0.001, yMax=pt2[1]+0.001, zMax=pt2[2]+0.001)
    try:
        framePart.RemoveWireEdges(wireEdgeList=wireEdges)
    except:
        return

# --------------------- Coloumn grid coordinates --------
# This function returns a list of all coloumn coordinates in XZ-plane
# REQUIRED ARGUMENTS:
# grid - list imported from excel file containing all coordinates used to draw beams and columns
def coloumn_coords(grid):
```

```
    colCoords = []
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for i in range(len(z_coord_lst)):
        z = z_coord_lst[i]
        for j in range(len(x_coord_matrix[i])):
            x = x_coord_matrix[i][j]
            colCoords.append((x,z))
    return colCoords

# -------------------- List of X-axes coordinates ---------------
# This function returns list of x-axes coordinates
# REQUIRED ARGUMENTS:
# grid - list imported from excel file containing all coordinates
    used to draw beams and columns
def x_axes_coords(grid):
    x_coord_lst = []
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for i in range(len(x_coord_matrix)):
        for j in range(len(x_coord_matrix[i])):
            if x_coord_matrix[i][j] not in x_coord_lst:
                 x_coord_lst.append(x_coord_matrix[i][j])
    x_coord_lst.sort()
    return x_coord_lst

# ------------ Functions combining other functions in order to
    create frame and floors -----------------
# The following set of functions combine previously defined
    functions in order to create the frame
def get_diag_indices(plane, diag_dict):
    ind_lst = []
    key_lst = []
    for key in diag_dict.keys():
        if diag_dict[key]['Plane'].lower() == plane.lower():
            ax_lst = diag_dict[key]['Axis']
            ind_lst += ax_lst
            if key not in key_lst:
                key_lst.append(key)
    return ind_lst, key_lst

def create_all_diagonals(framePart, diag_dict, connector_dict,
    grid):
    for key in diag_dict.keys():
        diag = diag_dict[key]
        diagSegLen = connector_dict[key][0]
```

```python
        ax_lst = diag['Axis']
        plane = diag['Plane']
        startAxis, endAxis = (diag['Start Column'], diag['End Column'])
        startLevel, endLevel = (diag['Start Level'], diag['End Level'])
        skipLevels = diag['Skip Levels']
        intersectAt = diag['Intersect At']
        for axis in ax_lst:
            beamDiagIntersect, colDiagIntersect = diagonal_intersections(plane, startAxis, endAxis, startLevel, endLevel, skipLevels, grid, axis, framePart, intersectAt)
            draw_diagonals(plane, framePart, colDiagIntersect, diagSegLen)


def create_all_frames(framePart, diag_dict, connector_dict, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    frames_diag_ind, diag_plane_keys = get_diag_indices('xy', diag_dict)
    frames_diag_ind = list(dict.fromkeys(frames_diag_ind))
    frames_no_diag_ind = list(range(len(z_coord_lst)))
    for i in frames_diag_ind:
        frames_no_diag_ind.remove(i)
    for z_ind in frames_no_diag_ind:
        if z_ind in [0, len(z_coord_lst)-1]:
            segmentLength = connector_dict['ShortEdgeBeams'][0]
        else:
            segmentLength = connector_dict['InnerBeams'][0]
        create_columns(framePart, grid, z_ind)
        create_beams('xy', framePart, grid, z_ind, segmentLength)
    for key in diag_plane_keys:
        diag = diag_dict[key]
        ax_lst = diag['Axis']
        plane = diag['Plane']
        startAxis, endAxis = (diag['Start Column'], diag['End Column'])
        startLevel, endLevel = (diag['Start Level'], diag['End Level'])
        skipLevels = diag['Skip Levels']
        intersectAt = diag['Intersect At']
        for z_ind in ax_lst:
```

```
            beamDiagIntersect, colDiagIntersect = diagonal_intersections(plane, startAxis, endAxis, startLevel, endLevel, skipLevels, grid, z_ind, framePart, intersectAt)
            if z_ind in [0, len(z_coord_lst)-1]:
                segmentLength = connector_dict['ShortEdgeBeams'][0]
            else:
                segmentLength = connector_dict['InnerBeams'][0]
            create_columns(framePart, grid, z_ind)
            create_beams_diag('xy', framePart, grid, z_ind, segmentLength, beamDiagIntersect)

def create_outer_beams(framePart, diag_dict, connector_dict, grid, floor_dict):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    beamLevels = get_beam_levels(floor_dict)
    segmentLength = connector_dict['LongEdgeBeams'][0]
    frames_diag_ind, diag_plane_keys = get_diag_indices('yz', diag_dict)
    for key in diag_plane_keys:
        diag = diag_dict[key]
        ax_lst = diag['Axis']
        for x_ind in ax_lst:
            plane = diag['Plane']
            startAxis, endAxis = (diag['Start Column'], diag['End Column'])
            startLevel, endLevel = (diag['Start Level'], diag['End Level'])
            skipLevels = diag['Skip Levels']
            intersectAt = diag['Intersect At']
            if x_ind in [0, len(x_coord_lst)-1]:
                beamDiagIntersect, colDiagIntersect = diagonal_intersections( plane, startAxis, endAxis, startLevel, endLevel, skipLevels, grid, x_ind, framePart, intersectAt)
                create_beams_diag('yz',framePart, grid, x_ind, segmentLength, beamDiagIntersect, beamLevels)

def build_frame(framePart, diag_dict, connector_dict, grid, beamLevels):
    create_all_diagonals(framePart, diag_dict, connector_dict, grid)
    create_all_frames(framePart, diag_dict, connector_dict, grid)
```

```python
    create_outer_beams(framePart, diag_dict, connector_dict, grid,
     ↪  beamLevels)

def build_floors(floorPart, start_level, end_level, grid):
    for level in range(start_level, end_level+1):
        create_floor(floorPart, grid, level)

# ------------- Get beam levels ----------------
# This function returns a list of the levels at which the outer
 ↪  beams
# should be created.
# REQUIRED ARGUMENTS:
# floor_dict - dictionary containging input of the different floors
def get_beam_levels(floor_dict):
    beamLevels = []
    for key in floor_dict.keys():
        include = floor_dict[key][4]
        if include:
            startLevel = floor_dict[key][0]
            endLevel = floor_dict[key][1]
            level = startLevel
            while level <= endLevel:
                beamLevels.append(level)
                level += 1
    beamLevels.sort()
    return beamLevels

# ------------- Remove Beams and Coloumns -------
# This function removes specified beams and coloumns from the frame
# REQUIRED ARGUMENTS:
# framePart - the part hosting the frame
# remove_dict - dictionary containg data on what beams and coloumns
 ↪  to be removed
# grid -  List/matrix imported from excel containing coordinates of
 ↪  the axis system
# OPTIONAL ARGUMENTS:
# tol - tolerance used to ensure that all desired objects are
 ↪  selected by bonding box
def remove_wires(framePart, remove_dict, grid, tol = 0.001):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    for key in remove_dict.keys():
        parts = remove_dict[key]['Parts']
        plane = remove_dict[key]['Plane']
```

```
        axis = remove_dict[key]['Axis']
        startLevel = remove_dict[key]['Start Level']
        endLevel = remove_dict[key]['End Level']
        startCol = remove_dict[key]['Start Column']
        endCol = remove_dict[key]['End Column']
        if parts == 'Columns' or parts == 'Beams and Columns':
            removeEdgeCols = remove_dict[key]['Remove Start/End']
        for i in axis:
            if plane == 'XY':
                xStart = x_coord_lst[startCol]
                xEnd = x_coord_lst[endCol]
                zStart = z_coord_lst[i]
                zEnd = z_coord_lst[i]
            elif plane == 'YZ':
                xStart = x_coord_lst[i]
                xEnd = x_coord_lst[i]
                zStart = z_coord_lst[startCol]
                zEnd = z_coord_lst[endCol]
            yStart = y_coord_lst[startLevel]
            yEnd = y_coord_lst[endLevel]

            pt1 = (xStart-tol, yStart-tol, zStart-tol)
            pt2 = (xEnd+tol, yEnd+tol, zEnd+tol)

            if parts == 'Beams and Columns' or parts == 'Beams':
                remove_edges_within_box(framePart, pt1, pt2,
                 ↪  setName='BeamSet')
            if parts == 'Beams and Columns' or parts == 'Columns':
                if removeEdgeCols == 0:
                    if plane == 'XY':
                        pt1 = (xStart+0.1, yStart-tol, zStart-tol)
                        pt2 = (xEnd-0.1, yEnd+tol, zEnd+tol)
                    elif plane == 'YZ':
                        pt1 = (xStart-tol, yStart-tol, zStart+0.1)
                        pt2 = (xEnd+tol, yEnd+tol, zEnd-0.1)
                remove_edges_within_box(framePart, pt1, pt2,
                 ↪  setName='ColumnSet')

# ------------------- Add Wires -----------------
# This function adds wires to existing part and saves them as
 ↪  separate sets
# The function also saves the connector parts of the wires to a set
# REQUIRED ARGUMENTS:
# framePart - part the wire should be added to
```

```
# add_dict - dictionary containing data about the wires that should
↪  be added
def add_wires(framePart, add_dicts, tol = 0.01):
    placements = add_dicts['Placement']
    sections = add_dicts['Section']
    orientations = add_dicts['Orientation']
    includeConn = add_dicts['IncludeConn']
    connectors = add_dicts['Connector']
    if len(placements) == 0:
        return
    startPts = []
    endPts = []
    for key in placements.keys():
        startPt = placements[key]['Start Point']
        endPt = placements[key]['End Point']
        dX = endPt[0]-startPt[0]
        dY = endPt[1]-startPt[1]
        dZ = endPt[2]-startPt[2]
        #Draw wire and save edges to unique sets
        if includeConn[key]:
            connectorLst = []
            segLength = connectors[key][0]
            totLength = np.sqrt(np.power(dX,2)+np.power(dY,2)+np.po
            ↪  wer(dZ,2))
            if totLength <= 2*segLength:
                segLength = totLength/2
            if startPt[0]!=endPt[0] and startPt[1]!=endPt[1] and
            ↪  startPt[2]!=endPt[2]:
                print('ERROR: Added wire "'+key+'" is not placed in
                ↪  one of the principal planes!')
            elif startPt[0] != endPt[0] and startPt[1] == endPt[1]
            ↪  and startPt[2] == endPt[2]:
                startSegPt = (startPt[0]+segLength, startPt[1],
                ↪  startPt[2])
                endSegPt = (endPt[0]-segLength, endPt[1], endPt[2])
            elif startPt[0] == endPt[0] and startPt[1] != endPt[1]
            ↪  and startPt[2] == endPt[2]:
                startSegPt = (startPt[0], startPt[1]+segLength,
                ↪  startPt[2])
                endSegPt = (endPt[0], endPt[1]-segLength, endPt[2])
            elif startPt[0] == endPt[0] and startPt[1] == endPt[1]
            ↪  and startPt[2] != endPt[2]:
                startSegPt = (startPt[0], startPt[1],
                ↪  startPt[2]+segLength)
```

```
                endSegPt = (endPt[0], endPt[1], endPt[2]-segLength)

            if isclose(totLength,2*segLength):
                wirePts = [startPt, startSegPt, endPt]
            else:
                wirePts = [startPt, startSegPt, endSegPt, endPt]
            for i in range(len(wirePts)-1):
                startPts.append(wirePts[i])
                endPts.append(wirePts[i+1])
        else:
            startPts.append(startPt)
            endPts.append(endPt)
    listOfWirePtsTuples = []
    for i in range(len(startPts)):
        listOfWirePtsTuples.append((startPts[i],endPts[i]))
    tupleOfWirePtsTuples = tuple(listOfWirePtsTuples)
    framePart.WirePolyLine(points=tupleOfWirePtsTuples)
```

C.8 TTB_post_processing.py

This file contains functions used to gather and process the results after a simulation. Used for getting the eigenfrequencies, calculating the damping ratio/logarithmic decrements and writing the results to a .txt file.

```
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import os
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys
import datetime
import step

sys.path.append(scriptsFolder)

from TTB_general import *
from TTB_geometry import *
from TTB_sets import *

# -------------- Extract Eigenfreqs -------------
# This function gets the natural frequencies from the .odb file and
↪ returns them as a list.
def get_eigenfreqs(jobName='TTBJob', stepName='FrequencyStep'):
    freqsLst = []
    odb = odbAccess.openOdb(jobName+'.odb')
```

```
    try:
        freqStep = odb.steps[stepName]
    except:
        print('Unable to find '+stepName+' in odb file. Check that the step is included in the Excel input file')
        return []
    region = freqStep.historyRegions['Assembly ASSEMBLY']
    try:
        freqs = region.historyOutputs['EIGFREQ'].data
    except:
        print('Unable to find EIGFREQ data for step: '+stepName)
    for i in range(len(freqs)):
        freqsLst.append(freqs[i][1])
    odb.close()
    return freqsLst


# -------------- Write to file -------------
# Writes list to file. Types: 'w+' overwrite excisting file, 'a+' append to file
# Option to write the indices before each row.
def list_to_file(list, fileName, type, indices=False, startIndex=0, indexStep=1, date_heading=True):
    f = open(fileName, type)
    if date_heading:
        d = get_date_and_time()
        f.write(d+'\n')
    i = startIndex
    for item in list:
        if indices:
            f.write(str(i)+'; '+str(item)+'\n')
        else:
            f.write(str(item)+'\n')
        i += indexStep
    f.close()

# Writes a dictonary to file, sorted by the keys.
# Types: 'w+' overwrite excisting file, 'a+' append to file
def dict_to_file(dictionary, fileName, type, date_heading=True):
    f = open(fileName, type)
    if date_heading:
        d = get_date_and_time()
        f.write(d+'\n')
    keysLst = dictionary.keys()
```

```
    keysLst.sort()
    for key in keysLst:
        val = dictionary[key]
        f.write(str(key)+'; '+str(val)+'\n')
    f.close()

# Writes item to file. Can be list, dict, int, float, bool etc.
# Option to create header with date and time of writing.
def write_to_file(item, fileName, type, print_date=True):
    try:
        item.keys() # To check if item is a dict.
        dict_to_file(item, fileName, type, date_heading=print_date)
        return
    except:
        pass
    try:
        iter(item) # To check if item is a list/tuple.
        list_to_file(item, fileName, type, date_heading=print_date)
        return
    except:
        pass
    try:
        f = open(fileName, type)
        if print_date:
            d = get_date_and_time()
            f.write(d+'\n')
        f.write(str(item)+'\n')
        f.close()
        return
    except:
        print('Could not write item to file...')


# -------------- Get peaks -------------
# Returns the magnitude and time values for the peaks of a time series.
# Simple algorithm, some filtering etc. should be added.
def get_peaks(dir, floorPart, var='U', outputNodeSetName='OUTPUTNODESET', jobName='TTBJob', stepName='FreeVibrationStep'):
    odb = odbAccess.openOdb(jobName+'.odb')
    floorPartOdb = odb.parts[floorPart.name.upper()]
    outputSet = odb.rootAssembly.instances[floorPart.name.upper()].nodeSets[outputNodeSetName]
```

```
    freeVibStep = odb.steps[stepName]
    freeVibFrames = freeVibStep.frames
    u_vec = []
    time_vec = []
    peak_lst = []
    if dir.lower() == 'x':
        ind = 0
    elif dir.lower() == 'y':
        ind = 1
    elif dir.lower() == 'z':
        ind = 2
    for i in range(len(freeVibFrames)):
        time_vec.append(float(freeVibFrames[i].frameValue))
        u_vec.append(float(freeVibFrames[i].fieldOutputs[var].getSubset(region=outputSet).values[0].data[ind]))
    for i in range(1,len(u_vec)-1):
        if (u_vec[i]>u_vec[i-1]) and (u_vec[i]>u_vec[i+1]) and (u_vec[i]>0):
            peak = (time_vec[i], u_vec[i])
            peak_lst.append(peak)
    odb.close()
    return peak_lst


# -------------- Calculate logarithmic decrement -------------
# Estimates the logarithmic decrement of a underdamped structure from a list
# containing the peaks of a time series (mag. and time). (Only positive peaks)
def log_dec(peak_lst, start_ind=1, n=2):
    t1, x1 = peak_lst[start_ind]
    t2, x2 = peak_lst[start_ind+n]
    if x2 >= x1:
        print('Log_Dec (Structural) - Warning: The value of the second peak are greater than or equal to to first peak. Log_dec (structural) is set to 0 (if Abaqus Based is choosen)')
        return 0
    ld = np.log(x1/x2)/n
    return ld


# -------------- Calculate natural frequency from peaks -------------
```

```python
# Estimates the natural frequency of a underdamped structure from a
↪  list
# containing the peaks of a time series. (mag. and time). (Only
↪  positive peaks)
def freq_from_peaks(peak_lst, start_ind=1, n=2):
    t1, x1 = peak_lst[start_ind]
    t2, x2 = peak_lst[start_ind+n]
    T = (t2-t1)/n
    freq = 1/T
    return freq


# -------------- Calculate damping ratio -------------
# Calculates the damping ratio based on a logarithmic decrement
↪  value.
# Assumes lightly damped structures.
def damping_ratio(logarithmic_decrement):
    if logarithmic_decrement == 0:
        return 0
    g = (1+(2*math.pi/logarithmic_decrement)**2)
    dr = 1/(g**0.5)
    return dr
```

C.9 TTB_properties.py

This file contains functions used to assign different properties to objects. Such properties include material data and cross sections.

```
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys
import step

sys.path.append(scriptsFolder)

from TTB_general import *
from TTB_sets import *
from TTB_geometry import *
from TTB_boundaries import *

# -------------- Create (rectangular) cross sections -------------
## Input: A dictionary with the cross section name as keys and a
↪ tuple with corresponding the dimensions (w*h) as value.
## Returns a tuple with the names of the created cross sections.
def create_cross_sections(crossSectionDict):
    model = get_model()
    lstOfNames = []
    for cs_name in crossSectionDict.keys():
```

```
        w = crossSectionDict[cs_name][0]
        h = crossSectionDict[cs_name][1]
        material_name = crossSectionDict[cs_name][2]
        model.RectangularProfile(name=cs_name, a=w, b=h)
        model.BeamSection(name=cs_name, profile=cs_name, integration=DURING_ANALYSIS, material=material_name)
        lstOfNames.append(cs_name)
    return tuple(lstOfNames)

# -------------- Create shell cross sections -------------
# This function creates the homogeous shell sections defined in sectionDict (imported from excel).
def create_shell_section(sectionDict):
    model = get_model()
    lstOfNames = []
    for cs_name in sectionDict.keys():
        t = sectionDict[cs_name][0]
        material_name = sectionDict[cs_name][1]
        model.HomogeneousShellSection(name=cs_name, material=material_name, thickness=t)

# -------------- Define materials -------------
## The following sets of functions defines different types of materials
## This function defines an orthotropic material
def create_ortho_material(matName, matDensity, E_1, E_2, E_3, Nu_12, Nu_13, Nu_23, G_12, G_13, G_23):
    model = get_model()
    material = model.Material(name=matName)
    material.Density(table=((matDensity, ),  ))
    material.Elastic(table=((E_1, E_2, E_3, Nu_12, Nu_13, Nu_23, G_12, G_13, G_23),  ), type = ENGINEERING_CONSTANTS)
    return material

## This function defines a transversely isotropic material
def create_trans_iso_material(matName, matDensity, E_1, E_2, Nu_12, Nu_23, G_12):
    material = create_ortho_material(matName, matDensity, E_1, E_2, E_2, Nu_12, Nu_12, Nu_23, G_12, G_12, E_2/(2*(1+Nu_23)))
    return material

## This function defines an isotropic material
def create_isotropic_material(matName, matDensity, E_1, Nu_12):
    model = get_model()
```

```python
    material = model.Material(name=matName)
    material.Density(table=((matDensity, ),))
    material.Elastic(table=((E_1,Nu_12), ))
    return material

## This function creates all the materials specified in the allMaterialsDict by the use of the functions above.
def create_material_from_dict(allMaterialsDict):
    model = get_model()
    material_names = allMaterialsDict.keys()
    for matName in material_names:
        matDict = allMaterialsDict[matName]
        type = matDict['Type']
        matDensity = matDict['Density']
        E_1 = matDict['E1']
        Nu_12 = matDict['Nu12']
        if type == 'Isotropic':
            create_isotropic_material(matName, matDensity, E_1, Nu_12)
        elif type in ['Trans. Isotropic', 'Orthotropic']:
            E_2 = matDict['E2']
            Nu_23 = matDict['Nu23']
            G_12 = matDict['G12']
            if type == 'Trans. Isotropic':
                create_trans_iso_material(matName, matDensity, E_1, E_2, Nu_12, Nu_23, G_12)
            elif type == 'Orthotropic':
                E_3 = matDict['E3']
                Nu_13 = matDict['Nu_13']
                G_13 = matDict['G13']
                G_23 = matDict['G23']
                create_ortho_material(matName, matDensity, E_1, E_2, E_3, Nu_12, Nu_13, Nu_23, G_12, G_13, G_23)

## This function adds the damping parameters from damping_dict to already created materials.
def add_material_damping(damping_dict):
    model = get_model()
    for mat_name in damping_dict.keys():
        mat = model.materials[mat_name]
        a, b, c, s = damping_dict[mat_name]
        mat.Damping(alpha=a, beta=b, composite=c, structural=s)
```

```python
# -------------- Create and assign beam-type cross sections -------------
## Every cross section name must have a matching Set (same name).
## This functions creates and assigns (beam) sections in crossSectionDict to the framePart.
def section_assignment(framePart, crossSectionDict, orientationsDict):
    create_cross_sections(crossSectionDict)
    for csName in crossSectionDict.keys():
        setName = csName
        edgesForCs = framePart.sets[setName].edges
        regionForCs = regionToolset.Region(edges=edgesForCs)
        orientationTuple = orientationsDict[csName]
        framePart.SectionAssignment(region=regionForCs, sectionName=csName)
        framePart.assignBeamSectionOrientation(region=regionForCs, method=N1_COSINES, n1=orientationTuple)


# ------- Section Assignemnt of Added Wires ------------
# REQUIRED ARGUMENTS:
# framePart - part hosting the added Wires
# add_dicts - dictionary containing data about the wires that is be added
def assign_section_added_wires(framePart, add_dicts):
    crossSectionsDict = add_dicts['Section']
    orientationsDict = add_dicts['Orientation']
    section_assignment(framePart, crossSectionsDict, orientationsDict)

# -------------- Create and assign cross sections -------------
## Every cross section name must have a matching Set (same name).
## This functions assigns shell sections in crossSectionDict to the floorPart.
## Can also be used for other parts containing shells.
def shell_section_assignment(floorPart, crossSectionDict):
    create_shell_section(crossSectionDict)
    for csName in crossSectionDict.keys():
        setName = csName
        try:
            facesForCs = floorPart.sets[setName].faces
            regionForCs = regionToolset.Region(faces=facesForCs)
            floorPart.SectionAssignment(region=regionForCs, sectionName=csName)
```

```
        except:
            continue

# ------- Section Assignemnt of Shafts ------------
# This functions assigns the shell sections to the shaft walls.
def shaft_section_assignment(shaftPart, sectionsWalls):
    create_shell_section(sectionsWalls)
    for csName in sectionsWalls.keys():
        if csName == 'Shaft Walls':
            facesForCs = shaftPart.faces
            regionForCs = regionToolset.Region(faces=facesForCs)
            shaftPart.SectionAssignment(region=regionForCs, sectionName=csName)

# ------- Section Assignemnt of Floors ------------
# This function creates subsets of the floorSet and assigns the respective sets with the correct cross sections.
def floor_assignment_from_dict(floorPart, floorSet, floor_dict, grid):
    model = get_model()
    for key in floor_dict.keys():
        start_level = floor_dict[key][0]
        end_level = floor_dict[key][1]
        t = floor_dict[key][2]
        mat_name = floor_dict[key][3]
        s = set_of_selected_floors(floorPart, floorSet, start_level, end_level, grid)
        model.HomogeneousShellSection(name=key, material=mat_name, thickness=t)
        reg = regionToolset.Region(faces=s.faces)
        floorPart.SectionAssignment(region=reg, sectionName=key)

# ------- Assigning sections of connector elements --------
## Connector Section Assignment
## This function can be used to create and assign connector segments defined by
## fractions of the original width and height of the member. (Currently not in use)
def connector_assignment_auto2(framePart, originalCrossSectionDict, connector_dict, tol):
    model = get_model()
    colSet = framePart.sets['ColumnSet']
    beamSet = framePart.sets['BeamSet']
    diagSet = framePart.sets['DiagonalSet']
```

```python
    for originalCS_Name in originalCrossSectionDict.keys():
        origSet = framePart.sets[originalCS_Name]
        newSetName = originalCS_Name+'_connectors'
        segmentLength = connector_dict[originalCS_Name][0]
        if 'beam' in newSetName.lower():
            s = set_of_connectors(framePart, newSetName, origSet, diagSet, colSet, segmentLength, tol)
        elif 'diag' in newSetName.lower():
            s = set_of_connectors(framePart, newSetName, origSet, colSet, False, segmentLength, tol)
        elif 'col' in newSetName.lower():
            s = set_of_connectors(framePart, newSetName, origSet, beamSet, diagSet, segmentLength, tol)
        else:
            print('Error in connector_assignment_auto')

        w_orig, h_orig, material_name = originalCrossSectionDict[originalCS_Name]
        w_ratio, h_ratio = connector_dict[originalCS_Name][1:]
        model.RectangularProfile(name=newSetName, a=w_orig*w_ratio, b=h_orig*h_ratio)
        model.BeamSection(name=newSetName, profile=newSetName, integration=DURING_ANALYSIS, material=material_name)

        edgesForCs = s.edges
        regionForCs = regionToolset.Region(edges=edgesForCs)
        framePart.SectionAssignment(region=regionForCs, sectionName=newSetName)

# ------- Assigning section for connector elements with generalized profile --------
## Connector Section Assignment
## This function can be used to create and assign connector segments defined by fractions of the original A, I11, I22 and J of the member.
def connector_assignment_auto_generalized_profile(framePart, originalCrossSectionDict, connector_dict, mat_dict, tol=0.001):
    model = get_model()
    colSet = framePart.sets['ColumnSet']
    beamSet = framePart.sets['BeamSet']
    diagSet = framePart.sets['DiagonalSet']
    for originalCS_Name in originalCrossSectionDict.keys():
        origSet = framePart.sets[originalCS_Name]
        newSetName = originalCS_Name+'_connectors'
```

```
        segmentLength = connector_dict[originalCS_Name][0]
        if 'beam' in newSetName.lower():
            s = set_of_connectors(framePart, newSetName, origSet, diagSet, colSet, segmentLength, tol)
        elif 'diag' in newSetName.lower():
            s = set_of_connectors(framePart, newSetName, origSet, colSet, False, segmentLength, tol)
        elif 'col' in newSetName.lower():
            s = set_of_connectors(framePart, newSetName, origSet, beamSet, diagSet, segmentLength, tol)
        else:
            print('Error in connector_assignment_auto')

        A_frac = connector_dict[originalCS_Name][1][0]
        A, I11, I22, J = connector_dict[originalCS_Name][2]
        alpha, beta, composite = connector_dict[originalCS_Name][3]
        mat_name = originalCrossSectionDict[originalCS_Name][2]
        youngsMod = mat_dict[mat_name]['E1']
        pois = mat_dict[mat_name]['Nu12']
        dens = (1/A_frac)*mat_dict[mat_name]['Density']
        try:
            shearMod = mat_dict[mat_name]['G12']
        except:
            shearMod = youngsMod/(2*(1+pois))
        model.GeneralizedProfile(name=newSetName, area=A, i11=I11, i22=I22, j=J, i12=0, gamma0=0, gammaW=0)
        model.BeamSection(name=newSetName, profile=newSetName, integration=BEFORE_ANALYSIS, poissonRatio=pois, density=dens, table=((youngsMod, shearMod),), alphaDamping=alpha, betaDamping=beta, compositeDamping=composite)

        edgesForCs = s.edges
        regionForCs = regionToolset.Region(edges=edgesForCs)
        framePart.SectionAssignment(region=regionForCs, sectionName=newSetName)

# ------- Exterior Wall Connector Zones ----------
# This function creates connector zones of the exterior walls and assignes the correct cross sections.
def walls_with_connectors_section_assignment_auto(wallPart, originalSectionDict, wallConnectorDict, grid):
    model = get_model()
    for val in wallConnectorDict.values():
```

```python
        field_width = val['Section'][0]
        try:
            if field_width != prev_field_width:
                field_width = prev_field_width
                print('Warning: Different field widths are unsuported, field with '+str(field_width)+'are used...')
        except:
            pass

    create_connector_panels_walls(wallPart, grid, field_width)
    for originalSection_name in originalSectionDict.keys():
        try:
            set = wallPart.sets[originalSection_name]
        except:
            continue

        inner_name = originalSection_name+'_center'
        outer_name = originalSection_name+'_connection'
        innerSet, outerSet = subsets_of_wall_panels(wallPart, set, inner_name, outer_name)
        orig_thickness, mat_name = originalSectionDict[originalSection_name]
        conn_mat_name = originalSection_name+' Connector Material'
        try:
            material = model.materials[conn_mat_name]
        except KeyError:
            conn_mat_name = wallConnectorDict[originalSection_name]['Section'][2]
        thickness_fraction = wallConnectorDict[originalSection_name]['Section'][1]
        model.HomogeneousShellSection(name=inner_name, material=mat_name, thickness=orig_thickness)
        inner_reg = regionToolset.Region(faces=innerSet.faces)
        wallPart.SectionAssignment(region=inner_reg, sectionName=inner_name)
        model.HomogeneousShellSection(name=outer_name, material=conn_mat_name, thickness=thickness_fraction*orig_thickness)
        outer_reg = regionToolset.Region(faces=outerSet.faces)
        wallPart.SectionAssignment(region=outer_reg, sectionName=outer_name)
```

```
# ------- Assign Section to Floor Connectors ----------
# Assigns cross sections to the connection zones of the floors.
def floor_connector_assignment(floorPart, floor_dict, shell_connector_dict):
    model = get_model()
    for key in floor_dict.keys():
        floor = floor_dict[key]
        if floor[5] == 1:
            connector = shell_connector_dict[key]
            name = key+'_Connectors'
            thickness_fraction = connector['Section'][1]
            matName = key+' Connector Material'
            origThickness = floor[2]
            model.HomogeneousShellSection(name=name, material=matName, thickness=origThickness*thickness_fraction)
            f = floorPart.sets[name].faces
            connReg = regionToolset.Region(faces=f)
            floorPart.SectionAssignment(region=connReg, sectionName=name)


# ---------- Assign Connector Sections to Added Wires --------
# This function assigns properties to the connectors of the added Wires
# REQUIRED ARGUMENT:
# framePart - part hosting the added wires
# add_dicts - dictionary containing data on the wires and the connectors
# mat-dict - material dictionary
def assign_connector_added_wire(framePart, add_dicts, mat_dict):
    model = get_model()
    crossSectionsDict = add_dicts['Section']
    orientationsDict = add_dicts['Orientation']
    connectorsDict = add_dicts['Connector']
    for key in connectorsDict.keys():
        connectorSetName = key+' Connectors'
        originalCS = crossSectionsDict[key]
        connector = connectorsDict[key]

        A_frac = connector[1][0]
        A, I11, I22, J = connector[2]
        alpha, beta, composite = connector[3]
        mat_name = originalCS[2]
```

```
        youngsMod = mat_dict[mat_name]['E1']
        pois = mat_dict[mat_name]['Nu12']
        dens = (1/A_frac)*mat_dict[mat_name]['Density']
        try:
            shearMod = mat_dict[mat_name]['G12']
        except:
            shearMod = youngsMod/(2*(1+pois))
        model.GeneralizedProfile(name=connectorSetName, area=A,
         ↪  i11=I11, i22=I22, j=J, i12=0, gamma0=0, gammaW=0)
        model.BeamSection(name=connectorSetName,
         ↪  profile=connectorSetName, integration=BEFORE_ANALYSIS,
         ↪  poissonRatio=pois, density=dens, table=((youngsMod,
         ↪  shearMod),), alphaDamping=alpha, betaDamping=beta,
         ↪  compositeDamping=composite)
        edgesForCs = framePart.sets[connectorSetName].edges
        regionForCs = regionToolset.Region(edges=edgesForCs)
        framePart.SectionAssignment(region=regionForCs,
         ↪  sectionName=connectorSetName)


# --------- Create Shell Connector Material ---------
# Creates a duplicate of the material assigned to the connection
 ↪  zones to allow for different damping properties from the
 ↪  original material
def shell_connector_material(material_dict, sectionsWalls,
 ↪  floor_dict, shell_connector_dict):
    model = get_model()
    for key in shell_connector_dict.keys():
        material_name = shell_connector_dict[key]['Section'][2]
        if key in sectionsWalls.keys():
            orig_mat_name = sectionsWalls[key][1]
        if key in floor_dict.keys():
            orig_mat_name = floor_dict[key][3]
        if material_name != orig_mat_name:
            orig_mat_name = material_name
        material_name = key+' Connector Material'
        orig_material = model.materials[orig_mat_name]
        material = model.Material(name=material_name,
         ↪  objectToCopy=orig_material)
        a, b, c, s = shell_connector_dict[key]['Damping']
        material.Damping(alpha=a, beta=b, composite=c, structural=s)


```

```python
# --------- Create Connector Material for Floor-to-shaft Connectors ---------
# Creates a duplicate of the material assigned to the connection zones to allow for different damping properties from the original material
def floor_to_shaft_material(material_dict, floor_dict, floor_to_shaft_dict):
    model = get_model()
    for key in floor_to_shaft_dict.keys():
        material_name = floor_to_shaft_dict[key]['Section'][2]
        orig_mat_name = floor_dict[key][3]
        if material_name != orig_mat_name:
            orig_mat_name = material_name
        material_name = key +' F-S Material'
        orig_material = model.materials[orig_mat_name]
        material = model.Material(name=material_name, objectToCopy=orig_material)
        a, b, c, s = floor_to_shaft_dict[key]['Damping']
        material.Damping(alpha=a, beta=b, composite=c, structural=s)


# --------- Create and Assign Section to Floor-to-shaft Connectors -------------
## Creates and assigns the modified sections for use in the connections between the floors and shafts.
def assign_floor_shaft_connector(floorPart, floor_dict, floor_to_shaft_dict):
    model = get_model()
    for floor_key in floor_to_shaft_dict.keys():
        floor = floor_dict[floor_key]
        connection = floor_to_shaft_dict[floor_key]
        connSection = connection['Section']
        connSetName = floor_key+' Floor-to-shaft Connectors'
        try:
            facesForCs = floorPart.sets[connSetName].faces
            regionForCs = regionToolset.Region(faces=facesForCs)
            t = connSection[1]*floor[2]
            mat_name = floor_key +' F-S Material'
            model.HomogeneousShellSection(name=connSetName, material=mat_name, thickness=t)
            floorPart.SectionAssignment(region=regionForCs, sectionName=connSetName)
        except KeyError:
            pass
```

```
# --------- Material orientation of the floors -------------
# Creates a local CSys and assigns the specified material orientations to the corresponding floors.
def orient_floors(floorPart, floorDict):
    for key in floorDict.keys():
        mat_dir = floorDict[key][7]
        try:
            mat_dir.lower()
        except:
            continue # Moves to next floor if orientation is not specified
        floorPart.DatumPointByCoordinate(coords=(0,0,0)) # Origin of local csys
        k = floorPart.datums.keys()
        if mat_dir.lower() == "x":
            floorPart.DatumPointByCoordinate(coords=(1,0,0)) # 1-Dir of local csys
            floorPart.DatumPointByCoordinate(coords=(1,0,1)) # Point in 1-2 plane of local csys
        elif mat_dir.lower() == "z":
            floorPart.DatumPointByCoordinate(coords=(0,0,1)) # 1-Dir of local csys
            floorPart.DatumPointByCoordinate(coords=(1,0,1)) # Point in 1-2 plane of local csys
        else:
            continue # Moves to next floor if orientation is not specified
        k = floorPart.datums.keys()
        k.sort()
        orig = floorPart.datums[k[-3]]
        local_1_dir_point = floorPart.datums[k[-2]]
        local_12_plane_point = floorPart.datums[k[-1]]
        floorPart.DatumCsysByThreePoints(name=key+"_mat_csys", coordSysType=CARTESIAN, origin=orig, point1=local_1_dir_point, point2=local_12_plane_point)
        k = floorPart.datums.keys()
        k.sort()
        matCsys = floorPart.datums[k[-1]]
        reg = floorPart.sets[key]
        floorPart.MaterialOrientation(region=reg, localCsys=matCsys, axis=AXIS_3, orientationType=SYSTEM)
        print('Material orientation added to '+key)
```

C.10 TTB_sets.py

This file contains all the functions related to creating sets of all kinds of objects in Abaqus e.g. beams, columns, surfaces etc...

```
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sketch
import part
import material
import section
import assembly
import mesh
import job
import odbAccess
import interaction
import load
import sys
import step

sys.path.append(scriptsFolder)

from TTB_general import *

from TTB_geometry import *

# -------------- Create Set of Diagonal Intersection Vertices
    -----------
# This function creates a set of all vertices where the diagonals
    intersects with other members
# REQUIRED ARGUMENTS:
# framePart - Part hosting the framePart
# beamDiagIntersectList - list of all beam-diagonal intersection
    Points
```

```python
# colDiagIntersectList - list of all beam-diagonal intersection Points
def diagonal_intersections_set(framePart, beamDiagIntersectList, colDiagIntersectList):
    bArray = []
    cArray = []
    for i in range(len(beamDiagIntersectList)):
        for j in range(len(beamDiagIntersectList[i])):
            bArray.append(framePart.vertices.findAt(((beamDiagIntersectList[i][j]),)))
            bds = framePart.Set(name="BeamDiagonalIntersectionSet", vertices=bArray)
    for i in range(len(colDiagIntersectList)):
        for j in range(len(colDiagIntersectList[i])):
            cArray.append(framePart.vertices.findAt(((colDiagIntersectList[i][j]),)))
            cds = framePart.Set(name="ColoumnDiagonalIntersectionSet", vertices=cArray)
    return (bds,cds)

# -------------- Create Sets Of All Columns, Beams, And Diagonals -------------
## Takes the frame part as input and returns a tuple of sets (ColumnSet, BeamSet, DiagonalSet)
def create_sets(framePart):
    allEdges = framePart.edges
    allVertices = framePart.vertices
    beamLst = []
    colLst = []
    diagLst = []
    xDirBeamLst = []
    for e in allEdges:
        verticeIDs = e.getVertices()
        sX, sY, sZ = allVertices[verticeIDs[0]].pointOn[0]
        eX, eY, eZ = allVertices[verticeIDs[1]].pointOn[0]
        if (sX == eX) and (sZ == eZ):
            colLst.append(e)
        elif sY == eY:
            beamLst.append(e)
            if sZ == eZ:
                xDirBeamLst.append(e)
        else:
            diagLst.append(e)
```

```python

    colLst = filter(None, colLst)
    colArray = part.EdgeArray(colLst)
    cs = framePart.Set(name='ColumnSet', edges=colArray)
    beamLst = filter(None, beamLst)
    beamArray = part.EdgeArray(beamLst)
    bs = framePart.Set(name='BeamSet', edges=beamArray)
    diagLst = filter(None, diagLst)
    diagArray = part.EdgeArray(diagLst)
    ds = framePart.Set(name='DiagonalSet', edges=diagArray)
    xDirBeamLst = filter(None, xDirBeamLst)
    XDirBeamArray = part.EdgeArray(xDirBeamLst)
    framePart.Set(name='XDirBeams', edges=XDirBeamArray)
    return (cs,bs,ds)


# -------------- List of vertices on edge -------------
# This function takes a set of edges, and returns a list of the vertices found on the edges
def get_vertices_from_edges(set_of_edges):
    edgesLst = set_of_edges.edges
    vertLst = []
    for e in edgesLst:
        vert = e.getVertices()
        for i in [0,1]:
            if vert[i] not in vertLst:
                vertLst.append(vert[i])
    return vertLst


# -------------- Create Set Of Short "Connector" Elements -------------
## This function creates set of beam-type connector elements
## REQUIRED ARGUMENTS:
## framePart - part hosting the connector elements
## newSetName - name of set to be created
## originSet - set the original member belong to
## set2 - Based on the use of the function the input should be:
##        set of all diagonals if set of beam connector elements should be created
##        set of all columns if set of diagonal connector elements should be created
##        set of all beams if set of column connector elemnts should be created
```

```python
## set3 - Based on the use of the function the input should be:
##        set of all columns if set of beam connector elements should be created
##        False if set of diagonal connector elements should be created
##        set of all diagonals if set of column connector elemnts should be created
## segmentLength - length of connector segment
## tol - geometric tolerance used when selecting the segments
## OPTIONAL ARGUMENTS:
## check - If true, a check if of the selected connector segments is conducted

def set_of_connectors(framePart, newSetName, originSet, set2, set3, segmentLength, tol, check=True):
    allVertices = framePart.vertices
    edgesLst = []
    edgesInSet = originSet.edges
    set2vert = get_vertices_from_edges(set2)
    set3vert = []
    if set3:
        set3vert = get_vertices_from_edges(set3)
    set2and3vert = set2vert + set3vert

    for i in range(len(edgesInSet)):
        if edgesInSet[i] not in edgesLst:
            beamSeq = edgesInSet[i:i+1]
            edLen = framePart.getLength(beamSeq)
            if edLen < segmentLength+tol:
                edgesLst.append(edgesInSet[i])

    edgesLst = filter(None, edgesLst)
    if check:
        for e in edgesLst:
            vert1 = e.getVertices()[0]
            vert2 = e.getVertices()[1]
            if (vert1 not in set2and3vert) and (vert2 not in set2and3vert):
                edgesLst.remove(e)

    edgesArray = part.EdgeArray(edgesLst)

    s = framePart.Set(name=newSetName, edges=edgesArray)
    return s
```

```python

# -------------- Create Set Of All Corner Columns -------------
## REQUIRED INPUT:
## framePart - part hosting the frame
## colSet - set containing all columns
## grid - List of lists containg the grid system (x,y and z coordinates)
def set_of_corner_cols(framePart, colSet, grid):
    edgesLst = []
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    y_min = y_coord_lst[0]
    y_max = y_coord_lst[-1]
    for i in [0,-1]:
        z = z_coord_lst[i]
        for j in [0,-1]:
            x = x_coord_matrix[i][j]
            edgesLst += colSet.edges.getByBoundingCylinder(center1=(x,y_min,z), center2=(x,y_max,z), radius=0.01)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    s = framePart.Set(name='CornerColumns', edges=edgesArray)
    return s

# -------------- Create Set Of All Outer Columns -------------
## REQUIRED ARGUMENTS:
## framePart - part hosting the frame
## colSet - set containing all columns
## xWidth - transverse width of building
## zWidth - longitudinal width of building
def set_of_outer_cols(framePart, colSet, xWidth, zWidth):
    edgesLst =  []
    for i in [0,1]:
            x = i*xWidth
            z = i*zWidth
            edgesLst += colSet.edges.getByBoundingBox(xMin=(x-0.01),zMin=-0.01, xMax=(x+0.01),zMax=(zWidth+0.01))
            edgesLst += colSet.edges.getByBoundingBox(xMin=-0.01,zMin=(z-0.01), xMax=(xWidth+0.01), zMax=(z+0.01))

    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
```

```
    s = framePart.Set(name='OuterColoumns', edges=edgesArray)
    return s

# -------------- Create Sets Of Edge Columns (-Corner) -------------
# Creates and returns sets of columns (Long Edge, Short Edge), excluding corner columns.
## REQUIRED ARGUMENTS:
## framePart - part hosting the frame
## colSet - set containing all columns
## cornerColSet - set of all corner columns
## xWidth - transverse width of building
## zWidth - longitudinal width of building
def sets_of_edge_cols(framePart, colSet, cornerColSet, xWidth, zWidth):
    cornerColumnsEdges = cornerColSet.edges
    edgesLst = []
    for i in [0,1]:
        x = i*xWidth
        edgesLst += colSet.edges.getByBoundingBox(xMin=x-0.01, zMin=0, xMax=x+0.01, zMax=zWidth)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    LE = framePart.Set(name='LongEdgeColumns', edges=edgesArray, xEdges=cornerColumnsEdges)

    edgesLst = []
    for i in [0,1]:
        z = i*zWidth
        edgesLst += colSet.edges.getByBoundingBox(xMin=0, zMin=z-0.01, xMax=xWidth, zMax=z+0.01)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    SE = framePart.Set(name='ShortEdgeColumns', edges=edgesArray, xEdges=cornerColumnsEdges)
    return (LE, SE)

# ---------------- Create set of Inner Columns ------
## REQUIRED ARGUMENTS:
## framePart - part hosting the frame
## colSet - set containing all columns
## cornerColSet - set of all corner columns
## longEdgeColSet - set of LongEdgeColumns
## shortEdgeColSet - set of ShortEdgeColumns
## xWidth - transverse width of building
```

```
## zWidth - longitudinal width of building
def set_of_inner_cols(framePart, colSet, cornerColSet, longEdgeColSet, shortEdgeColSet, xWidth, zWidth):
    excludeColumnsEdges = cornerColSet.edges + longEdgeColSet.edges + shortEdgeColSet.edges
    edgesLst = []
    edgesLst += colSet.edges.getByBoundingBox(xMin=0, zMin=0, xMax=xWidth, zMax=zWidth)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    IC = framePart.Set(name='InnerColumns', edges=edgesArray, xEdges=excludeColumnsEdges)
    return IC


# -------------- Create all sets of columns -------------
## Combines the previously defined function, and creates all subsets of columns
## REQUIRED ARGUMENTS:
## framePart - part hosting the frame
## colSet - set containing all columns
## grid - List of lists containg the grid system (x,y and z coordinates)
def sets_of_cols(framePart, colSet, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    xWidth = max(x_axes_coords(grid))
    zWidth = max(z_coord_lst)
    cornerColumnsSet = set_of_corner_cols(framePart,colSet, grid)
    longEdgeColumnsSet, shortEdgeColumnsSet = sets_of_edge_cols(framePart, colSet, cornerColumnsSet, xWidth, zWidth)
    innerColumnsSet = set_of_inner_cols(framePart, colSet, cornerColumnsSet, longEdgeColumnsSet, shortEdgeColumnsSet, xWidth, zWidth)
    outerColoumnSet = set_of_outer_cols(framePart, colSet, xWidth, zWidth)

    ## Checking
    nC = len(colSet.edges)
    nCC = len(cornerColumnsSet.edges)
    nLE = len(longEdgeColumnsSet.edges)
    nSE = len(shortEdgeColumnsSet.edges)
    nI = len(innerColumnsSet.edges)

```

```python
    if nC != (nCC+nLE+nSE+nI):
        print('WARNING: Total number of columns are different from the number placed in subsets.')

    return (cornerColumnsSet, longEdgeColumnsSet, shortEdgeColumnsSet, innerColumnsSet)


# --------------- Sets of edge beams ------------
## Creates sets of LongEdgeBeams and Short Edge Beams
## REQUIRED ARGUMENTS:
## framePart - part hosting the frame
## beamSet - set containing all beams
## grid - List of lists containg the grid system (x,y and z coordinates)
## xWidth - transverse width of building
## zWidth - longitudinal width of building
## height - height of building
def sets_of_edge_beams(framePart, beamSet, xWidth, zWidth, height):
    edgesLst = []
    for i in [0,1]:
            x = i*xWidth
            edgesLst += beamSet.edges.getByBoundingBox(xMin=x-0.01, yMin=0, zMin=0, xMax=x+0.01, yMax=height, zMax=zWidth)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    LE = framePart.Set(name='LongEdgeBeams', edges=edgesArray)

    edgesLst = []
    for i in [0,1]:
            z = i*zWidth
            edgesLst += beamSet.edges.getByBoundingBox(xMin=0, yMin=0, zMin=z-0.01, xMax=xWidth, yMax=height, zMax=z+0.01)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    SE = framePart.Set(name='ShortEdgeBeams', edges=edgesArray)
    return (LE, SE)


# --------------- Sets of inner beams ------------
## Creates set of internal beams
## REQUIRED ARGUMENTS:
```

```
## framePart - part hosting the frame
## beamSet - set containing all beams
## longEdgeBeamSet - set containing all LongEdgeBeams
## shortEdgeBeamSet - set containing all ShortEdgeBeams
## xWidth - transverse width of building
## zWidth - longitudinal width of building
## height - height of building
def set_of_inner_beams(framePart, beamSet, longEdgeBeamSet,
 ↪ shortEdgeBeamSet, xWidth, zWidth, height):
    excludeBeamEdges = longEdgeBeamSet.edges +
     ↪ shortEdgeBeamSet.edges
    edgesLst = []
    edgesLst += beamSet.edges.getByBoundingBox(xMin=0, yMin=0,
     ↪ zMin=0, xMax=xWidth, yMax=height, zMax=zWidth)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    IB = framePart.Set(name='InnerBeams', edges=edgesArray,
     ↪ xEdges=excludeBeamEdges)
    return IB

# -------------- Create all sets of beams -------------
## Combines the previously defined functions and creates all
 ↪ subsets of beams
## REQUIRED ARGUMENTs:
## framePart - part hosting the frame
## beamSet - set containing all beams
## grid - List of lists containg the grid system (x,y and z
 ↪ coordinates)
def sets_of_beams(framePart, beamSet, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    xWidth = max(x_axes_coords(grid))
    zWidth = max(z_coord_lst)
    height = max(y_coord_lst)
    longEdgeBeamSet, shortEdgeBeamSet =
     ↪ sets_of_edge_beams(framePart, beamSet, xWidth, zWidth,
     ↪ height)
    innerBeamSet = set_of_inner_beams(framePart, beamSet,
     ↪ longEdgeBeamSet, shortEdgeBeamSet, xWidth, zWidth, height)

    ## Checking
    nB = len(beamSet.edges)
    nLE = len(longEdgeBeamSet.edges)
    nSE = len(shortEdgeBeamSet.edges)
    nI = len(innerBeamSet.edges)
```

```

    if nB != (nLE+nSE+nI):
        print('WARNING: Total number of beams are different from
         ↪  the number placed in subsets.')

    return (longEdgeBeamSet, shortEdgeBeamSet, innerBeamSet)


# -------------- Create all sets of diagonals -------------
## Creates set of LongdEdgeDiagonals and ShortEdgeDiagonals
## REQUIRED ARGUMENTs:
## framePart - part hosting the frame
## diagonalSet - set containing all diagonals
## grid - List of lists containg the grid system (x,y and z
 ↪  coordinates)
def sets_of_diagonals(framePart, diagonalSet, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    xWidth = max(x_axes_coords(grid))
    zWidth = max(z_coord_lst)
    height = max(y_coord_lst)
    edgesLst = []
    for i in [0,1]:
            x = i*xWidth
            edgesLst +=
             ↪  diagonalSet.edges.getByBoundingBox(xMin=x-0.01,
             ↪  yMin=0, zMin=0, xMax=x+0.01, yMax=height,
             ↪  zMax=zWidth)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    LE = framePart.Set(name='LongEdgeDiagonals', edges=edgesArray)

    edgesLst = []
    for i in [0,1]:
            z = i*zWidth
            edgesLst += diagonalSet.edges.getByBoundingBox(xMin=0,
             ↪  yMin=0, zMin=z-0.01, xMax=xWidth, yMax=height,
             ↪  zMax=z+0.01)
    edgesLst = filter(None, edgesLst)
    edgesArray = part.EdgeArray(edgesLst)
    SE = framePart.Set(name='ShortEdgeDiagonals', edges=edgesArray)
    return (LE, SE)

# -------------- Create set of all walls -------------
## REQUIRED ARGUMENTS:
```

```
## wallPart - part hosting walls
def create_set_all_walls(wallPart):
    f = wallPart.faces
    ws = wallPart.Set(name='Walls', faces=f)
    return ws

# -------------- Create subsets of wall panels -------------
## Creates sets of Outer (connectors) and Inner (original wall) Wall panel
## REQUIRED ARGUMENTS:
## wallPart - part hosting walls
## allWallsSet - set of all walls
## inner_name - name of inner set
## outer_name - name of outer set
def subsets_of_wall_panels(wallPart, allWallsSet, inner_name, outer_name):
    allFaces = allWallsSet.faces
    innerList = []
    outerList = []
    for f in allFaces:
        vertList = f.getVertices()
        if len(vertList) == 4:
            innerList.append(f)
        else:
            outerList.append(f)
    innerArray = part.FaceArray(innerList)
    innerSet = wallPart.Set(name='CenterPanels', faces=innerArray)
    outerArray = part.FaceArray(outerList)
    outerSet = wallPart.Set(name='WallPanelConnectors', faces=outerArray)
    return innerSet, outerSet

# -------------- Create set of all floors -------------
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
def create_set_all_floors(floorPart):
    f = floorPart.faces
    FS = floorPart.Set(name='Floors', faces=f)
    return FS

# -------------- Create set of selected floors -------------
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
## allFloorsSet - set of all floors
```

```
## fromLevel - index of lowest level to be included in set
## toLevel - index of hihgest level to be included in set
## grid - List of lists containg the grid system (x,y and z coordinates)
def set_of_selected_floors(floorPart, allFloorsSet, fromLevel, toLevel, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    setName = 'Floors_'+str(fromLevel)+'-'+str(toLevel)
    yStart = y_coord_lst[fromLevel]
    yEnd = y_coord_lst[toLevel]+0.001
    f = allFloorsSet.faces.getByBoundingBox(yMin=yStart-0.01, yMax=yEnd+0.01)
    FS = floorPart.Set(name=setName, faces=f)
    return FS

# -------------- Create surface of bottom surface of floors -------------
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
## grid - List of lists containg the grid system (x,y and z coordinates)
def surface_of_bottom_floor(floorPart, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    surfName = 'Slab Surface'
    yStart = y_coord_lst[0]
    yEnd = y_coord_lst[0]
    f = floorPart.faces.getByBoundingBox(yMin=yStart-0.01, yMax=yEnd+0.01)
    floorPart.Surface(name=surfName, side1Faces=f)

# -------------- Creates set of floor types from dictionary -------------
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
## floor_dict - dictionary containing information on floors
## grid - List of lists containg the grid system (x,y and z coordinates)
def set_of_floor_types(floorPart, floor_dict, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for key in floor_dict.keys():
        setName = key
        floor = floor_dict[key]
        fromLevel = floor[0]
        toLevel = floor[1]
```

```
        yStart = y_coord_lst[fromLevel]-0.001
        yEnd = y_coord_lst[toLevel]+0.001
        f = floorPart.faces.getByBoundingBox(yMin=yStart, yMax=yEnd)
        FS = floorPart.Set(name=setName, faces=f)
    return FS

# -------------- Creates set of selected walls -------------
## REQUIRED ARGUMENTS:
## wallPart - part hosting walls
## allWallsSet - set of all walls
## planePos - grid line index defining position of walls
## plane - 'xy' or 'yz'
def set_of_selected_walls(wallPart, allWallsSet, planePos, plane):
    if plane.lower() == 'xy':
        setName = 'xyWall_z='+str(planePos)
        f = allWallsSet.faces.getByBoundingBox(zMin=planePos-0.001, zMax=planePos+0.001)
    elif plane.lower() == 'yz':
        setName = 'yzWall_x='+str(planePos)
        f = allWallsSet.faces.getByBoundingBox(xMin=planePos-0.001, xMax=planePos+0.001)
    else:
        print('Error in wall set creation, wrong plane definition.')
    ws = wallPart.Set(name=setName, faces=f)
    return ws

# -------------- Creates set of all shafts -------------
## REQUIRED ARGUMENTS:
## shaftPart - part hosting shafts
def set_of_all_shafts(shaftPart):
    f = shaftPart.faces
    if f:
        SS = shaftPart.Set(name='Shaft Walls', faces=f)
        return SS

# -------------- Create individual sets of all shafts -------
## REQUIRED ARGUMENTS:
## shaftPart - part hosting shafts
## shaft_dict - dictionary containing information about shafts
## grid - List of lists containg the grid system (x,y and z coordinates)
def set_of_single_shaft(shaftPart, shaft_dict, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for key in shaft_dict.keys():
```

```
        shaft = shaft_dict[key]
        if shaft['Connect To Building']:
            pt1_xz = tuple(shaft['Start Coordinate'])
            pt2_xz = tuple(shaft['End Coordinate'])
            startLevel = shaft['Start Level']
            endLevel = shaft['End Level']
            endLevelOffset = shaft['End Level Offset']
            removeWall = shaft['Remove Wall']
            yStart = y_coord_lst[startLevel]
            yEnd = y_coord_lst[endLevel]+ endLevelOffset

            pt1 = (pt1_xz[0], yStart, pt1_xz[1])
            pt2 = (pt2_xz[0], yEnd, pt2_xz[1])
            f = shaftPart.faces.getByBoundingBox(xMin=pt1[0]-0.001, yMin=pt1[1]-0.001, zMin=pt1[2]-0.001, xMax=pt2[0]+0.001, yMax=pt2[1]+0.001, zMax=pt2[2]+0.001)
            shaftPart.Set(name=str(key), faces=f)
            shaftPart.Surface(name=str(key)+'_surface', side2Faces=f)


# ---------- Create set of floor edges that intercepts with shaft -------------
# This function creates a set of all floor edges adjacent to each shaft
# REQUIRED ARGUMENTS:
# floorPart - part containing the floors the set will be saved to
# shaft_dict - dictionary containg all relevant information regarding shaft geometry (generated from input file)
# grid - List of lists containg the grid system (x,y and z coordinates)
def sets_of_shaft_floor_edges(floorPart, shaft_dict, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    allSets = []
    for key in shaft_dict.keys():
        shaft = shaft_dict[key]
        if shaft['Connect To Building']:
            pt1_xz = tuple(shaft['Start Coordinate'])
            pt2_xz = tuple(shaft['End Coordinate'])
            startLevel = shaft['Start Level']
            endLevel = shaft['End Level']
            removeWall = shaft['Remove Wall']
            yStart = y_coord_lst[startLevel]
```

```
            yEnd = y_coord_lst[endLevel]

            pt1 = (pt1_xz[0], yStart, pt1_xz[1])
            pt2 = (pt2_xz[0], yEnd, pt2_xz[1])
            edgesLst =floorPart.edges.getByBoundingBox(xMin=pt1[0]-0.001, yMin=pt1[1]-0.001, zMin=pt1[2]-0.001, xMax=pt2[0]+0.001, yMax=pt2[1]+0.001, zMax=pt2[2]+0.001)
            floorPart.Set(name='FloorEdgesAround'+str(key), edges=edgesLst)
            allSets.append(floorPart.sets['FloorEdgesAround'+str(key)])
    if allSets:
        floorPart.SetByBoolean(name='AllFloorEdgesAroundShafts', operation=UNION, sets=allSets)

# --------- Shaft Surfaces to Tie to Floor ---------------
## Creates surface of outer surfacce of
## shaft that will be used for creating ties to floors
## REQUIRED ARGUMENTS:
## shaftPart - part hosting shafts
## floorPart - part hosting floors
## shaft_dict - dictionary containing information about shafts
## grid - List of lists containg the grid system (x,y and z coordinates)
def shaft_surfaces_for_ties(shaftPart, floorPart, shaft_dict, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for key in shaft_dict.keys():
        shaft = shaft_dict[key]
        if shaft['Connect To Building']:
            pt1_xz = tuple(shaft['Start Coordinate'])
            pt2_xz = tuple(shaft['End Coordinate'])
            startLevel = shaft['Start Level']
            endLevel = shaft['End Level']
            removeWall = shaft['Remove Wall']
            yStart = y_coord_lst[startLevel]
            yEnd = y_coord_lst[endLevel]
            pt1 = (pt1_xz[0], yStart, pt1_xz[1])
            pt2 = (pt2_xz[0], yEnd, pt2_xz[1])

# --------- Set of Floor Connectors ---------------
## Creates set of connector zones between floor elements
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
```

```
## floor_dict - dictionary containing information about floors
## shell_connector_dict - dictionary containing information
↪  shell_type connections
## grid - List of lists containg the grid system (x,y and z
↪  coordinates)
## OPTIONAL ARGUMENT:
## tol - tolerance for selecting the geometry
def set_of_floor_connectors(floorPart, floor_dict,
↪  shell_connector_dict, grid, tol = 0.001):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    xWidth = abs(x_coord_lst[-1]-x_coord_lst[0])
    for key in floor_dict.keys():
        floor = floor_dict[key]
        faceLst = []
        if floor[5] == 1:
            connector = shell_connector_dict[key]
            section = connector['Section']
            connWidth = section[0]
            approxElemWidth = floor[6]
            numOfConn = int(xWidth/approxElemWidth)+1
            elemWidth = xWidth/(numOfConn+1)
            setName = key
            xCoord = elemWidth
            while xCoord < x_coord_lst[-1]-connWidth:
                xmin = xCoord-connWidth/2-tol
                xmax = xCoord+connWidth/2+tol
                f = floorPart.sets[setName].faces.getByBoundingBox(
                ↪  xMin=xmin,
                ↪  xMax=xmax)
                for i in range(len(f)):
                    faceLst.append(f[i])
                xCoord += elemWidth
            faceArray = part.FaceArray(faceLst)
            floorPart.Set(faces=faceArray, name=key+'_Connectors')

# -------------- Create set contaning a single node (to be used to
↪  get results) -------------
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
## grid - List of lists containg the grid system (x,y and z
↪  coordinates)
## OPTIONAL ARGUMENTS:
## setName - name of set
```

```
## relX - x-position of node, as fraction of total x-width
## relY - y-position of node, as fraction of total y-width
## relZ - z-position of node, as fraction of total z-width
## If neither are altered, central node on top floor is selected
def create_output_node_set(floorPart, grid,
↪   setName='OUTPUTNODESET', relX=0.5, relY=1, relZ=0.5):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    x_coord = relX*(x_coord_lst[-1]-x_coord_lst[0])
    z_coord = relZ*(z_coord_lst[-1]-z_coord_lst[0])
    y_coord = relY*(y_coord_lst[-1]-y_coord_lst[0])
    n = floorPart.nodes
    outputNodeLst = [n.getClosest((x_coord, y_coord, z_coord))]
    outputNodeArray = mesh.MeshNodeArray(nodes=outputNodeLst)
    s = floorPart.Set(name=setName, nodes=outputNodeArray)
    return s

# ------------- Set of Outer Floor Edges ------
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
## grid - List of lists containg the grid system (x,y and z
↪   coordinates)
## OPTIONAL ARGUMENT:
## tol - tolerance for selecting the geometry
def outer_floor_edges_set(floorPart, grid, tol = 0.01):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    xmin=x_coord_lst[0]
    xmax=x_coord_lst[-1]
    zmin=z_coord_lst[0]
    zmax=z_coord_lst[-1]
    e_z = []
    e_x = []
    e_z.append(floorPart.edges.getByBoundingBox(xMin=xmin-tol,
    ↪   zMin=zmin-tol, xMax=xmin+tol, zMax=zmax+tol))
    e_z.append(floorPart.edges.getByBoundingBox(xMin=xmax-tol,
    ↪   zMin=zmin-tol, xMax=xmax+tol, zMax=zmax+tol))
    floorPart.Set(name='OuterFloorEdgesZDir', edges=e_z)
    e_x.append(floorPart.edges.getByBoundingBox(xMin=xmin-tol,
    ↪   zMin=zmin-tol, xMax=xmax+tol, zMax=zmin+tol))
    e_x.append(floorPart.edges.getByBoundingBox(xMin=xmin-tol,
    ↪   zMin=zmax-tol, xMax=xmax+tol, zMax=zmax+tol))
    floorPart.Set(name='OuterFloorEdgesXDir', edges=e_x)
    e = e_z + e_x
```

```
    floorPart.Set(name='OuterFloorEdges', edges=e)


# ------------ Floor Surfaces for Frame Ties -----------
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
def floor_surfaces(floorPart):
    f = floorPart.faces
    floorPart.Surface(name='FloorSurfaces', side1Faces=f)


# ------------ Set of Shaft Edges At Removed Wall ------------
## REQUIRED ARGUMENTS:
## shaftPart - part hosting shafts
## shaft_dict - dictionary containing information about shafts
## grid - List of lists containg the grid system (x,y and z
↪   coordinates)
## OPTIONAL ARGUMENT:
## tol - tolerance for selecting the geometry
def shaft_side_edges_set(shaftPart, shaft_dict, grid, tol=0.01):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    for key in shaft_dict.keys():
        shaft = shaft_dict[key]
        if shaft['Connect To Building']:
            pt1_xz = tuple(shaft['Start Coordinate'])
            pt2_xz = tuple(shaft['End Coordinate'])
            startLevel = shaft['Start Level']
            endLevel = shaft['End Level']
            endLevelOffset = shaft['End Level Offset']
            removeWall = shaft['Remove Wall']
            yStart = y_coord_lst[startLevel]
            yEnd = y_coord_lst[endLevel]+endLevelOffset

            if removeWall == 1:
                e = shaftPart.edges.getByBoundingBox(xMin=pt1_xz[0]-
                ↪   tol, yMin=yStart-tol, zMin=pt1_xz[1]-tol,
                ↪   xMax=pt2_xz[0]+tol, yMax=yEnd+tol,
                ↪   zMax=pt1_xz[1]+tol)
                shaftPart.Set(name=key+'_SideEdges', edges=e)

            if removeWall == 2:
```

```python
                e = shaftPart.edges.getByBoundingBox(xMin=pt2_xz[0]-tol, yMin=yStart-tol, zMin=pt1_xz[1]-tol, xMax=pt2_xz[0]+tol, yMax=yEnd+tol, zMax=pt2_xz[1]+tol)
                shaftPart.Set(name=key+'_SideEdges', edges=e)

            if removeWall == 3:
                e = shaftPart.edges.getByBoundingBox(xMin=pt1_xz[0]-tol, yMin=yStart-tol, zMin=pt2_xz[1]-tol, xMax=pt2_xz[0]+tol, yMax=yEnd+tol, zMax=pt2_xz[1]+tol)
                shaftPart.Set(name=key+'_SideEdges', edges=e)

            if removeWall == 4:
                e = shaftPart.edges.getByBoundingBox(xMin=pt1_xz[0]-tol, yMin=yStart-tol, zMin=pt1_xz[1]-tol, xMax=pt1_xz[0]+tol, yMax=yEnd+tol, zMax=pt2_xz[1]+tol)
                shaftPart.Set(name=key+'_SideEdges', edges=e)


# ----------- Set of Shaft Edges for Wall Ties ------------
## REQUIRED ARGUMENTS:
## shaftPart - part hosting shafts
## shaft_dict - dictionary containing information about shafts
## grid - List of lists containg the grid system (x,y and z coordinates)
def shaft_edges_for_wall_ties(shaftPart, shaft_dict, grid):
    shaft_side_edges_set(shaftPart, shaft_dict, grid)
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    setList = []
    for key in shaft_dict.keys():
        shaft = shaft_dict[key]
        if shaft['Connect To Building']:
            pt1_xz = tuple(shaft['Start Coordinate'])
            pt2_xz = tuple(shaft['End Coordinate'])
            startLevel = shaft['Start Level']
            endLevel = shaft['End Level']
            endLevelOffset = shaft['End Level Offset']
            removeWall = shaft['Remove Wall']
            yStart = y_coord_lst[startLevel]
            yEnd = y_coord_lst[endLevel]+endLevelOffset
```

```
            if isclose(pt1_xz[1],z_coord_lst[0],rel_tol=1e-06) and removeWall==1:
                setList.append(shaftPart.sets[key+'_SideEdges'])
            if isclose(pt2_xz[0],x_coord_lst[-1],rel_tol=1e-06) and removeWall==2:
                setList.append(shaftPart.sets[key+'_SideEdges'])
            if isclose(pt2_xz[0],z_coord_lst[-1],rel_tol=1e-06) and removeWall==3:
                setList.append(shaftPart.sets[key+'_SideEdges'])
            if isclose(pt1_xz[0],x_coord_lst[0],rel_tol=1e-06) and removeWall==4:
                setList.append(shaftPart.sets[key+'_SideEdges'])
    if setList:
        shaftPart.SetByBoolean(name='ShaftEdgesForWallTies', sets=setList, operation=UNION)


# ------------ Assembly Set of Edges Used For Wall TIes ---------
## REQUIRED ARGUMENTS:
## shaftPart - part hosting shafts
## framePart - part hosting frame
## floorPart - part hosting floors
## shellConnectorDict - dictionary containing information about shell connections
def edges_for_wall_ties_set(shaftPart, framePart, floorPart, shellConnectorDict):
    a = get_assembly()
    connectWallsTo = shellConnectorDict['Walls']['ConnectTo'].lower()
    ofe = a.allInstances[floorPart.name].sets['OuterFloorEdges']
    leb = a.allInstances[framePart.name].sets['LongEdgeBeams']
    seb = a.allInstances[framePart.name].sets['ShortEdgeBeams']
    oc = a.allInstances[framePart.name].sets['OuterColoumns']
    setList = []
    if 'floors' in connectWallsTo:
        setList.append(ofe)
    if 'beams' in connectWallsTo:
        setList.append(leb)
        setList.append(seb)
    if 'columns' in connectWallsTo:
        setList.append(oc)
    if len(setList) < 0:
        print('Error: Check wall connection input.')
```

```
    a.SetByBoolean(name='EdgesForWallTies', sets=setList, operation=UNION)


# ----------- Inner Surface of Wall ---------
## REQUIRED ARGUMENTS:
## wallPart- part hosting walls
def wall_surfaces(wallPart):
    f = wallPart.faces
    wallPart.Surface(name='InnerSurface', side1Faces=f)
    wallPart.Surface(name='OuterSurface', side2Faces=f)

# ----------- Set of Added Wire ----------
# This function creates individual sets of the added Wires
# REQUIRED ARGUMENTS:
# framePart - part hosting the added Wires
# add_dicts - dictionary containing required data about the added Wires
# OPTIONAL ARGUMENT:
# tol - tolerance used to ensure that all desired objects are selected by bonding box
#       default value is 0.01
def sets_of_added_wires(framePart, add_dicts, tol=0.01):
    placements = add_dicts['Placement']
    connectors = add_dicts['Connector']
    includeConn = add_dicts['IncludeConn']
    # Set of each individual wire
    for key in placements.keys():
        startPt = placements[key]['Start Point']
        endPt = placements[key]['End Point']
        dX = endPt[0]-startPt[0]
        dY = endPt[1]-startPt[1]
        dZ = endPt[2]-startPt[2]
        wireEdge = framePart.edges.getByBoundingBox(xMin=min(startPt[0],endPt[0])-tol, yMin=min(startPt[1],endPt[1])-tol, zMin=min(startPt[2],endPt[2])-tol, xMax=max(startPt[0],endPt[0])+tol, yMax=max(startPt[1],endPt[1])+tol, zMax=max(startPt[2],endPt[2])+tol)
        framePart.Set(name=key, edges=wireEdge)

        if includeConn[key]:
            segLength = connectors[key][0]
```

```
totLength = np.sqrt(np.power(dX,2)+np.power(dY,2)+np.power(dZ,2))
if totLength <= 2*segLength:
    connectorEdges = framePart.sets[key].edges
    framePart.Set(name=key+' Connectors', edges=connectorEdges)
else:
    if startPt[0]!=endPt[0] and startPt[1]!=endPt[1] and startPt[2]!=endPt[2]:
        print('ERROR: Added wire "'+key+'" is not placed in one of the principal planes!')
    elif startPt[0] != endPt[0] and startPt[1] == endPt[1] and startPt[2] == endPt[2]:
        startSegPt = (startPt[0]+segLength, startPt[1], startPt[2])
        endSegPt = (endPt[0]-segLength, endPt[1], endPt[2])
    elif startPt[0] == endPt[0] and startPt[1] != endPt[1] and startPt[2] == endPt[2]:
        startSegPt = (startPt[0], startPt[1]+segLength, startPt[2])
        endSegPt = (endPt[0], endPt[1]-segLength, endPt[2])
    elif startPt[0] == endPt[0] and startPt[1] == endPt[1] and startPt[2] != endPt[2]:
        startSegPt = (startPt[0], startPt[1], startPt[2]+segLength)
        endSegPt = (endPt[0], endPt[1], endPt[2]-segLength)

    startConnEdge = framePart.sets[key].edges.getByBoundingBox(xMin=min(startPt[0],startSegPt[0])-tol, yMin=min(startPt[1],startSegPt[1])-tol,zMin=min(startPt[2],startSegPt[2])-tol,xMax=max(startPt[0],startSegPt[0])+tol,yMax=max(startPt[1],startSegPt[1])+tol,zMax=max(startPt[2],startSegPt[2])+tol)
    endConnEdge = framePart.sets[key].edges.getByBoundingBox(xMin=min(endPt[0],endSegPt[0])-tol,yMin=min(endPt[1],endSegPt[1])-tol,zMin=min(endPt[2],endSegPt[2])-tol,xMax=max(endPt[0],endSegPt[0])+tol,yMax=max(endPt[1],endSegPt[1])+tol,zMax=max(endPt[2],endSegPt[2])+tol)
```

```
                framePart.Set(name=key+' Connectors', edges =
                ↪  [startConnEdge, endConnEdge])

# ------------- Floor-to-shaft connector set -------------
## Creates set of floor-to-shaft connector zones
## REQUIRED ARGUMENTS:
## floorPart - part hosting floors
## floor_dict - dictionary containing information about floors
## shaft_dict - dictionary containing information about shafts
## floor_to_shaft_dict - dictionary containing information about
↪  floor-to-shaft connections
## grid - List of lists containg the grid system (x,y and z
↪  coordinates)
# OPTIONAL ARGUMENT:
# tol - tolerance used to ensure that all desired objects are
↪  selected by bonding box default value is 0.01
def floor_to_shaft_set(floorPart, floor_dict, shaft_dict,
↪  floor_to_shaft_dict, grid, tol = 0.001):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    if floor_to_shaft_dict:
        for floor_key in floor_to_shaft_dict.keys():
            setName = floor_key+' Floor-to-shaft Connectors'
            floor = floor_dict[floor_key]
            floor_shaft_conn = floor_to_shaft_dict[floor_key]
            startLevel_floor = floor[0]
            endLevel_floor = floor[1]
            connWidth = floor_shaft_conn['Section'][0]
            faceLst = []
            for shaft_key in shaft_dict.keys():
                shaft = shaft_dict[shaft_key]
                if shaft['Connect To Building']:
                    startLevel_shaft = shaft['Start Level']
                    endLevel_shaft = shaft['End Level']
                    if startLevel_shaft < startLevel_floor:
                        startLevel = startLevel_floor
                    else:
                        startLevel = startLevel_shaft
                    if endLevel_shaft < endLevel_floor:
                        endLevel = endLevel_shaft
                    else:
                        endLevel = endLevel_floor
                    yStart = y_coord_lst[startLevel]
                    yEnd = y_coord_lst[endLevel]
```

```
                    #if startLevel_shaft == 0:
                    #    yStart = yStart+2*tol

                    xzStart_shaft = shaft['Start Coordinate']
                    xzEnd_shaft = shaft['End Coordinate']

                    xzStart_connector =
                     ↪  (xzStart_shaft[0]-connWidth,
                     ↪  xzStart_shaft[1]-connWidth)
                    xzEnd_connector = (xzEnd_shaft[0]+connWidth,
                     ↪  xzEnd_shaft[1]+connWidth)

                    f = floorPart.sets[floor_key].faces.getByBoundi⌋
                     ↪  ngBox(xMin=xzStart_connector[0]-tol,
                     ↪  yMin=yStart-tol,
                     ↪  zMin=xzStart_connector[1]-tol,
                     ↪  xMax=xzEnd_connector[0]+tol, yMax=yEnd+tol,
                     ↪  zMax=xzEnd_connector[1]+tol)
                    for i in range(len(f)):
                        faceLst.append(f[i])
                else:
                    continue
            faceArray = part.FaceArray(faceLst)
            if faceArray:
                floorPart.Set(faces=faceArray, name=setName)

# ------------------ Set of Nodes At Bottom of Columns ------------
## REQUIRED ARGUMENTS:
## framePart - part hosting frame
## grid - List of lists containg the grid system (x,y and z
 ↪  coordinates)
# OPTIONAL ARGUMENT:
# tol - tolerance used to ensure that all desired objects are
 ↪  selected by bonding box default value is 0.01
def set_of_bottom_nodes(framePart, grid, tol = 0.01):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    verts = framePart.vertices.getByBoundingBox(yMin=y_coord_lst[0]-
     ↪  tol,
     ↪  yMax=y_coord_lst[0]+tol)
    framePart.Set(vertices=verts, name='Column Ends')
```

C.11 TTB_Windload_EC.py

This file contains all the functions and formulas for calculating the wind load according to Eurocode 1.

```python
# Script for wind calculations according to Eurocode.
# Equation references are from NS-EN-1991-1-4 (inc. Appendices) unless otherwise is stated.
# Friction forces are assumed to be negligible.
# Forces on internal faces cancel each other (forces with equal magnitude acts on opposing faces).
# -------------- Input folder path -------------
# Folder where all the scripts are located:
scriptsFolder = 'C:\\Users\\username\\TTBParametricModel'

# -------------- Import Packages -------------
from abaqus import *
from abaqusConstants import *
import regionToolset
import numpy as np
import math
import sys
import sketch
import part
import material
import section
import assembly
import material
import mesh
import time
import odbAccess
import load
import random
import os
import step

sys.path.append(scriptsFolder)

from TTB_geometry import *
from TTB_excel import *
from TTB_post_processing import *
from TTB_general import *

```

```python
# Creates a dictionary containing all the parameters used in the calculations based on the excel imported dict and the results of the free vibration step.
def create_wind_param_dict(xlsx_dict, res_dict, grid):
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    d = {}
    cat = xlsx_dict['TerrainCat']
    d['z_0'], d['z_min'] = terrain_param(cat)

    for key in ['Cdir', 'Calt', 'Cseason', 'Co', 'kl', 'ModeExponent', 'v_b0',
                'WindDir', 'ReturnPeriod_Acc', 'ReturnPeriod_Load', 'SampleHeigth_Acc']:
        d[key] = xlsx_dict[key]

    d['AnnualExceedenceProb_Load'] = annual_exceedence_probability(d['ReturnPeriod_Load'])
    d['Cprob_Load'] = c_prob(d['AnnualExceedenceProb_Load'])

    d['AnnualExceedenceProb_Acc'] = annual_exceedence_probability(d['ReturnPeriod_Acc'])
    d['Cprob_Acc'] = c_prob(d['AnnualExceedenceProb_Acc'])

    if d['WindDir'].lower() == 'x':
        d['b'] = z_coord_lst[-1]-z_coord_lst[0]
        d['d'] = x_coord_lst[-1]-x_coord_lst[0]
    else:
        d['d'] = z_coord_lst[-1]-z_coord_lst[0]
        d['b'] = x_coord_lst[-1]-x_coord_lst[0]

    d['h'] = y_coord_lst[-1]
    d['r'] = xlsx_dict['r']
    d['z_s'] = max(0.6*d['h'], d['z_min'])

    if xlsx_dict['NatFreq'] == 'Abaqus':
        d['NatFreq'] = res_dict['NatFreq']
    elif xlsx_dict['NatFreq'] == 'Eurocode':
        d['NatFreq'] = 46/h
    else:
        d['NatFreq'] = xlsx_dict['NatFreq']

    if xlsx_dict['LogDec_Struct'] == 'Abaqus':
        d['LogDec_Struct'] = res_dict['LogDec_Struct']
```

```
    else:
        d['LogDec_Struct'] = xlsx_dict['LogDec_Struct']

    d['DampingRatio_Struct'] = damping_ratio(d['LogDec_Struct'])

    if xlsx_dict['LogDec_Aero'] == 'Eurocode':
        d['LogDec_Aero'] = delta_a(d)
    else:
        d['LogDec_Aero'] = xlsx_dict['LogDec_Aero']

    d['DampingRatio_Aero'] = damping_ratio(d['LogDec_Aero'])

    d['LogDec_Total'] = d['LogDec_Struct'] + d['LogDec_Aero']
    d['DampingRatio_Total'] = damping_ratio(d['LogDec_Total'])

    d['m_e'] = m_e(d)

    return d


# Change Cprob depending on type of calculation and its specified return periods (Acceleration vs Load)
def set_Cprob(set_to, param_dict):
    if set_to == None or set_to.lower() == 'none':
        param_dict['Cprob'] = None
        print('Cprob is set to "None".')
    elif set_to.lower() in ['acc', 'acceleration']:
        param_dict['Cprob'] = param_dict['Cprob_Acc']
        print('Cprob is set to "acceleration" value.')
    elif set_to.lower() in ['load']:
        param_dict['Cprob'] = param_dict['Cprob_Load']
        print('Cprob is set to "load" value.')
    elif set_to.lower() in ['delete', 'del']:
        try:
            del param_dict['Cprob']
            print('Cprob reset/deleted.')
        except:
            pass
    else:
        print('Error: Could not change C_prob (Invalid input)')
        set_Cprob('delete', param_dict)


# Terrain parameters (Table NA.4.1)
```

```python
def terrain_param(cat):
    if cat == 0:
        z_0 = 0.003
        z_min = 2.0
    elif cat == 1:
        z_0 = 0.01
        z_min = 2.0
    elif cat == 2:
        z_0 = 0.05
        z_min = 4.0
    elif cat == 3:
        z_0 = 0.3
        z_min = 8.0
    elif cat == 4:
        z_0 = 1.0
        z_min = 16.0
    else:
        print('Error: Invalid terrain category. Parameters for cat.
        ↪  2 set')
        return terrain_param(2)
    return z_0, z_min


# Basic wind velocity (Eq. NA.4.1)
def v_b(param_dict):
    c_dir = param_dict['Cdir']
    c_season = param_dict['Cseason']
    c_alt = param_dict['Calt']
    c_prob = param_dict['Cprob']
    v_b0 = param_dict['v_b0']
    return c_dir*c_season*c_alt*c_prob*v_b0


# Turbulence Length Scale (Eq. B.1)
def L(z, param_dict):
    z_0 = param_dict['z_0']
    z_min = param_dict['z_min']
    z_t = 200  # Ref. height
    L_t = 300  # Ref. length scale
    alpha = 0.67 + 0.05 * math.log(z_0)
    if z < z_min:
        return L(z_min, param_dict)
    else:
        return L_t*(z/z_t)**alpha
```

```python


# Terrain roughness coefficient (Eq. 4.5)
def k_r(param_dict):
    z_0 = param_dict['z_0']
    return 0.19*(z_0/0.05)**0.07


# Roughness Coefficient (Eq. 4.4)
def c_r(z, param_dict):
    z_0 = param_dict['z_0']
    z_min = param_dict['z_min']
    z_max = 200
    if z < z_min:
        return c_r(z_min, param_dict)
    elif z > z_max:
        return c_r(z_max, param_dict)
    else:
        return k_r(param_dict)*math.log(z/z_0)


# Annual exceedence probability
# Often the formula p=1/T is used, but it does not work with short
↪ return periods, therefore the exponetial expression
↪ p=1-exp(-1/T) is used.
def annual_exceedence_probability(return_period):
    T = return_period
    return 1-exp(-1/T)


# Propability Coefficient
def c_prob(p, K=0.2, n=0.5):
    f = 1-K*np.log(-np.log(1-p))
    g = 1-K*np.log(-np.log(0.98))
    return (f/g)**n


# Mean wind velocity (Eq. 4.3)
def v_m(z, param_dict):
    c_o = param_dict['Co']
    return c_r(z, param_dict)*c_o*v_b(param_dict)


# Non-dimensional frequency (Eq. B.2)
```

```
def f_L(z,n, param_dict):
    return n*L(z, param_dict)/v_m(z, param_dict)


# Non-dimensional power spectral density function (Eq. B.2)
def S_L(z,n, param_dict):
    s = (6.8*f_L(z,n,param_dict))/((1+10.2*f_L(z,n,param_dict))**(5
    ↪  /3))
    return s


# Background factor (Eq. B.3)
def B(param_dict):
    z_s = param_dict['z_s']
    b = param_dict['b']
    h = param_dict['h']
    g = ((b+h)/L(z_s,param_dict))**0.63
    return (1/(1+0.9*g))**0.5


# Eq. B.7
def eta_h(param_dict):
    z_s = param_dict['z_s']
    h = param_dict['h']
    n_1 = param_dict['NatFreq']
    return (4.6*h/L(z_s,param_dict))*f_L(z_s, n_1,param_dict)


# Eq. B.8
def eta_b(param_dict):
    z_s = param_dict['z_s']
    b = param_dict['b']
    n_1 = param_dict['NatFreq']
    return (4.6*b/L(z_s,param_dict))*f_L(z_s,n_1,param_dict)


# Aerodynamic admittance (Eq. B.7)
def R_h(param_dict):
    n = eta_h(param_dict)
    if n == 0:
        return 1
    else:
        return (1/n)-(1/(2*n**2))*(1-math.e**(-2*n))
```

```

# Aerodynamic admittance (Eq. B.8)
def R_b(param_dict):
    n = eta_b(param_dict)
    if n == 0:
        return 1
    else:
        return (1/n)-(1/(2*n**2))*(1-math.e**(-2*n))


# Resonance response factor (Eq. B.6)
def R(param_dict):
    log_dec = param_dict['LogDec_Total']
    z_s = param_dict['z_s']
    n_1 = param_dict['NatFreq']
    g = (math.pi**2/(2*log_dec))*S_L(z_s, n_1,param_dict)*R_h(param_dict)*R_b(param_dict)
    return g**0.5


# Equivalent mass (Section F.4(2))
def m_e(param_dict):
    h = param_dict['h']
    h_min = (2/3)*h
    m = get_model()
    part_keys = m.parts.keys()
    massUpperThird = 0
    for part_key in part_keys:
        prt = m.parts[part_key]
        edgeSelection = prt.edges.getByBoundingBox(yMin=h_min)
        faceSelection = prt.faces.getByBoundingBox(yMin=h_min)
        cellSelection = prt.cells.getByBoundingBox(yMin=h_min)
        edgeSelectionMass = prt.getMassProperties(regions=edgeSelection)['mass']
        faceSelectionMass = prt.getMassProperties(regions=faceSelection)['mass']
        cellSelectionMass = prt.getMassProperties(regions=cellSelection)['mass']

        if edgeSelectionMass:
            massUpperThird += edgeSelectionMass
        if faceSelectionMass:
            massUpperThird += faceSelectionMass
        if cellSelectionMass:
```

```
                massUpperThird += cellSelectionMass

        avgDistMassUpperThird = massUpperThird/(h/3)
        return avgDistMassUpperThird


# logarithmic Decrement (Aerodynamic) (Eq. F.16)
def delta_a(param_dict, acc_or_load='Load'):
    set_Cprob(acc_or_load, param_dict)
    h_vector, disp_vector = mode_shape_vector(param_dict)
    zs = param_dict['z_s']
    b = param_dict['b']
    n1 = param_dict['NatFreq']
    ro = 1.25
    cf = c_f(param_dict)
    vm = v_m(zs, param_dict)
    me = m_e(param_dict)
    da = (cf*ro*b*vm)/(2*n1*me)
    set_Cprob('delete', param_dict)
    return da


# Up-crossing frequency (Eq. B.5)
def nu(param_dict):
    n_1 = param_dict['NatFreq']
    v = n_1*(R(param_dict)**2/(B(param_dict)**2+R(param_dict)**2))*
    ↪   *0.5
    return max(v, 0.08)


# Peak Factor (Eq. B.4)
def k_p(param_dict, v=None):
    if not v:
        v = nu(param_dict)
    T = 600
    g = 2*math.log(v*T)
    if g <= 0:
        return 3
    else:
        return max(g**0.5+(0.6/(g**0.5)), 3)


# Standard deviation of turbulence (Eq. 4.6)
def sigma_v(param_dict):
```

```
    k_l = param_dict['kl']
    return k_r(param_dict)*v_b(param_dict)*k_l


# Turbulence intensity (Eq. 4.7)
def I_v(z, param_dict):
    z_max = 200
    z_min = param_dict['z_min']
    if z < z_min:
        return I_v(z_min, param_dict)
    elif z > z_max:
        return I_v(z_max, param_dict)
    else:
        return sigma_v(param_dict)/v_m(z, param_dict)


# Size Factor (Eq. 6.2)
def c_s(param_dict):
    z_s = param_dict['z_s']
    g = 7*I_v(z_s, param_dict)
    return (1+g*B(param_dict))/(1+g)


# Dynamic Factor (Eq. 6.3)
def c_d(param_dict):
    z_s = param_dict['z_s']
    f = 1+2*k_p(param_dict)*I_v(z_s,param_dict)*((B(param_dict)**2+R(param_dict)**2)**0.5)
    g = 1+7*I_v(z_s,param_dict)*B(param_dict)
    return f/g


# Peak velocity pressure (Eq. NA.4.8)
def q_p(z, param_dict):
    ro = 1.25
    qm = 0.5*ro*v_m(z, param_dict)**2
    qp = qm*(1+2*3.5*I_v(z, param_dict))
    return qp


# Exposure coefficient (Eq. 4.9)
def c_e(z, param_dict):
    return q_p(z, param_dict)/q_b(param_dict)
```

```python

# Basic velocity pressure (Eq. 4.10)
def q_b(param_dict):
    ro = 1.25
    return 0.5*ro*v_b(param_dict)**2


# Reference Height (Fig. 7.4)
def z_e(param_dict):
    h = param_dict['h']
    b = param_dict['b']
    if h <= b:
        return [h]
    elif h > 2*b:
        z_strip = (h-2*b)/4
        temp = [b+z_strip*i for i in range(5)]
        return temp+[h]
    else:
        return [b, h]


# Force coefficient of rectangular sections (Fig. 7.23)
# Using linear interpolation to approx. fig. 7.36 gives slightly inaccurate results...
def c_f0(param_dict):
    d = param_dict['d']
    b = param_dict['b']
    xp = [0.1, 0.2, 0.6, 0.7, 1, 2, 5, 10, 20, 50]
    fp = [2, 2, 2.35, 2.4, 2.1, 1.65, 1, 0.9, 0.9, 0.9]
    x = d/b
    return np.interp(x, xp, fp, left=None, right=None)


# Reduction factor for quadratic sections with rounded corners (Fig. 7.24)
def psi_r(param_dict):
    r = param_dict['r']
    b = param_dict['b']
    xp = [0, 0.2, 0.4]
    fp = [1, 0.5, 0.5]
    x = r/b
    return np.interp(x, xp, fp)

```

```
# Reduction factor for end effects (Tab. 7.16 + Fig. 7.36)
# Using linear interpolation to approx. fig. 7.36 gives slightly inaccurate results...
def psi_lambda(param_dict):
    h = param_dict['h']
    b = param_dict['b']
    l = h
    xp = [15, 50]
    fp = [1.0, 0.7]
    c = np.interp(l, xp, fp)
    lam = max(70, c*l/b)
    xp = [1, 10, 70, 200]
    fp = [0.6, 0.7, 0.92, 1]    # Assumes phi = 1.0 (Eq. 7.28)
    return np.interp(lam, xp, fp)


# Force coef. for struc. elements with rect. cross sections (Eq. 7.9)
def c_f(param_dict):
    return c_f0(param_dict)*psi_r(param_dict)*psi_lambda(param_dict)


# Factor for reduction in correlation (Section 7.2.2(3))
def c_corr(param_dict):
    h = param_dict['h']
    d = param_dict['d']
    x = h/d
    xp = [1, 5]
    fp = [0.85, 1]
    return np.interp(x, xp, fp)


# Shape factors
def c_pe10(param_dict):
    h = param_dict['h']
    d = param_dict['d']
    if h/d > 5:     # Use section 7.6 of Eurocode
        return c_corr(param_dict)*c_f(param_dict)
    else:           # Use section 7.2.2 of Eurocode
        x = h/d
        xp = [0.25, 1, 5]
        fp_D = [0.7, 0.8, 0.8]
        fp_E = [0.3, 0.5, 0.7]
        c_pe10D = np.interp(x, xp, fp_D)
```

```python
        c_pe10E = np.interp(x, xp, fp_E)
        return c_corr(param_dict)*(c_pe10D+c_pe10E)


# Wind pressure acting on exterior faces (eq. 5.1)
def w_e(param_dict):
    w = []
    for ze in z_e(param_dict):
        w.append(q_p(ze, param_dict)*c_pe10(param_dict))
    return w


# Exterior wind forces (Eq. 5.6). By default the force on a 1m^2
    face is calculated. But other areas can be specified.
# The area input can be a vector containing different areas for the
    different height division (see fig. 7.4), or a float if the
    area is the same for all zones (or if force by area is wanted).
def F_we(param_dict, A_ref=1):
    c = c_s(param_dict)*c_d(param_dict)
    cA = np.multiply(c, A_ref)
    return np.multiply(cA, w_e(param_dict))


# Non-dimensional coefficient (Eq. B.11)
def K_x(param_dict):
    zs = param_dict['z_s']
    z_vec, phi_vec = mode_shape_vector(param_dict)
    integrand_1 = np.zeros(len(z_vec))
    integrand_2 = np.zeros(len(z_vec))
    for i in range(len(integrand_1)):
        z = z_vec[i]
        integrand_1[i] = (v_m(z,param_dict)**2)*phi_vec[i]
        integrand_2[i] = phi_vec[i]**2
    f = np.trapz(y=integrand_1, x=z_vec)
    g = (v_m(zs, param_dict)**2)*np.trapz(y=integrand_2, x=z_vec)
    return f/g

# Standard deviation of the acceleration (Eq. B10)
# NB! Needs to be adjusted to correct return period!
def sigma_a(z, param_dict):
    set_Cprob('Acceleration', param_dict)
    z_vec, phi_vec = mode_shape_vector(param_dict)
    phi_z = np.interp(z, z_vec, phi_vec)
    ro = 1.25
```

```
    b = param_dict['b']
    zs = param_dict['z_s']
    cf = c_f(param_dict)
    Iv = I_v(zs, param_dict)
    Vm = v_m(zs, param_dict)
    m1 = m_e(param_dict)
    stdev =
    ↪ ((cf*ro*b*Iv*Vm**2)/m1)*R(param_dict)*K_x(param_dict)*phi_z
    set_Cprob('delete', param_dict)
    return stdev


# Peak (max) acceleration (Eq. B10)
def peak_acc(z, param_dict):
    set_Cprob('Acceleration', param_dict)
    natFreq = param_dict['NatFreq']
    val = k_p(param_dict, v=natFreq)*sigma_a(z, param_dict)
    set_Cprob('delete', param_dict)
    return val

# Apply the calculated force to the structure.
# If adjust_to_grid=True the height of the different horizonal
↪ strips are adjusted to mach the level heights.
# If this is not done the load will not be applied to the columns
↪ intersected by a change between strips.
# See fig. 7.4 in the Eurocode for what is meant by a "horizontnal
↪ strip"
def apply_EC_wind_force(frame_part, column_set, grid, param_dict,
↪ step_name='Static_Wind_Eurocode', adjust_to_grid=True):
    set_Cprob('Load', param_dict)
    counter = 0
    model = get_model()
    a = get_assembly()
    x_coord_matrix, y_coord_lst, z_coord_lst = grid
    x_coord_lst = x_axes_coords(grid)
    wind_dir = param_dict['WindDir']
    e = column_set.edges
    frame_inst = a.instances[frame_part.name]
    ze_vector = z_e(param_dict)
    ze_vector = [0]+ze_vector
    Fwe_vec = F_we(param_dict, A_ref=1)
    if wind_dir.lower() == 'x':
        x = x_coord_lst[-1]
        z_min = z_coord_lst[0]
```

```
        z_max = z_coord_lst[-1]
        for i in range(1, len(ze_vector)):
            y_below = ze_vector[i-1]
            y_current = ze_vector[i]
            if adjust_to_grid:
                y_below = closest(y_coord_lst, y_below)
                y_current = closest(y_coord_lst, y_current)
            for j in range(len(z_coord_lst)):
                z = z_coord_lst[j]
                load_width = get_load_width(j, z_coord_lst)
                load_mag = Fwe_vec[i-1]*load_width
                cols_for_load = frame_inst.edges.getByBoundingBox(xMin=x-0.01, xMax=x+0.01, yMin=y_below, yMax=y_current, zMin=z-0.01, zMax=z+0.01)
                reg_for_load = regionToolset.Region(edges=cols_for_load)
                load_name = 'WindLoad_'+str(counter)
                model.LineLoad(name=load_name, createStepName=step_name, region=reg_for_load, comp1=load_mag)
                counter += 1
    elif wind_dir.lower() == 'z':
        x_coord_lst = x_coord_matrix[0]
        z = z_coord_lst[0]
        x_min = x_coord_lst[0]
        x_max = x_coord_lst[-1]
        for i in range(1, len(ze_vector)):
            y_below = ze_vector[i-1]
            y_current = ze_vector[i]
            if adjust_to_grid:
                y_below = closest(y_coord_lst, y_below)
                y_current = closest(y_coord_lst, y_current)
            for j in range(len(x_coord_lst)):
                x = x_coord_lst[j]
                load_width = get_load_width(j, x_coord_lst)
                load_mag = Fwe_vec[i-1]*load_width
                cols_for_load = frame_inst.edges.getByBoundingBox(xMin=x-0.01, xMax=x+0.01, yMin=y_below, yMax=y_current, zMin=z-0.01, zMax=z+0.01)
                reg_for_load = regionToolset.Region(edges=cols_for_load)
                load_name = 'WindLoad_'+str(counter)
```

```
                model.LineLoad(name=load_name, createStepName=step_name, region=reg_for_load, comp3=load_mag)
                counter += 1
    set_Cprob('delete', param_dict)


# Calculte the load width. Used for converting pressure to line loads.
def get_load_width(ind, coordinate_vector):
    if ind == 0:
        start_ind = 0
        end_ind = ind+1
    elif ind == len(coordinate_vector)-1:
        start_ind = ind-1
        end_ind = ind
    else:
        start_ind = ind-1
        end_ind = ind+1
    coord = coordinate_vector[ind]
    start_coord = coordinate_vector[start_ind]
    end_coord = coordinate_vector[end_ind]
    width_a = 0.5*(coord-start_coord)
    width_b = 0.5*(end_coord-coord)
    return width_a + width_b


# Generate a mode shape vactor based on the input in the wind-sheet in the excel file.
def mode_shape_vector(param_dict, n_points=200):
    mode_exp = param_dict['ModeExponent']
    h = param_dict['h']
    if mode_exp == 'Abaqus':
        print('Abaqus mode shape is not yet implemented, it must be specified directly...')
        ## Not Finished!
    else:
        zeta = mode_exp
        h_vector = np.linspace(0, h, n_points)
        disp_vector = [(z/h)**zeta for z in h_vector]
    return h_vector, disp_vector
```

```python
# Creates a dictionary containing the results from the free
    vibration step, plus some basic information.
def free_vib_res_dict(xlsx_dict, floorPart):
    d = {}
    freeVib_direction = xlsx_dict['WindDir']
    peak_lst = get_peaks(freeVib_direction, floorPart)
    struct_log_dec = log_dec(peak_lst)
    struct_damp_ratio = damping_ratio(struct_log_dec)
    d['LogDec_Struct'] = struct_log_dec
    d['DampingRatio_Struct'] = struct_damp_ratio
    d['NatFreq'] = freq_from_peaks(peak_lst)
    return d


# Creates a dictionary containing the results from the accelartion
    response calculation, plus some basic information.
# The sampleHeigth option gives the option yo override the sample
    height specified in the excel file.
def acc_res_dict_EC(param_dict, sampleHeigth=None):
    d = {}
    if sampleHeigth: ## Possible to override excel input
        height_coordinate = sampleHeigth
        print('Warning: Override of Excel sample height is
            specified directly in script.')
    else:
        height_coordinate = param_dict['SampleHeigth_Acc']

    pA = peak_acc(height_coordinate, param_dict)
    stDev = sigma_a(height_coordinate, param_dict)
    d['1. Direction'] = param_dict['WindDir']
    d['2. Return Period'] = param_dict['ReturnPeriod_Acc']
    d['3. Height Coordinate'] = height_coordinate
    d['4. Natural Frequency'] = param_dict['NatFreq']
    d['5. Peak Acceleration'] = pA
    d['6. Standard Deviation (Acceleration)'] = stDev
    d['7. Peak Factor'] = pA/stDev
    set_Cprob('acc', param_dict)
    z_s = param_dict['z_s']
    d['8.1 v_b0'] = param_dict['v_b0']
    d['8.2 v_b'] = v_b(param_dict)
    d['8.3 v_m (At height z_s)'] = v_m(z_s, param_dict)
    set_Cprob('del', param_dict)
    return d
```

D. H. Reed and L. H. Wiig

NTNU
Norwegian University of
Science and Technology