Identification of structural properties of high-rise buildings

Application of a model updating technique for estimating the structural properties of high-rise buildings.

Master thesis

Isa Sophie Ritfeld

Delft University of Technology

Identification of structural properties of high-rise buildings

Application of a model updating technique for estimating the structural properties of high-rise buildings.

by

Isa Sophie Ritfeld

Student Name	I.S. Ritfeld
Student nummer	4892534

Supervisors of TU Delft:	Dr.ir. K.N. van Dalen and Dr. E. Lourens
Supervisors of TNO:	Ir. A.J. Bronkhorst and Ir .D. Moretti
Supervisor of Aronsohn:	Ir. P. Lagendijk
Project Duration:	October, 2023 - July, 2024
Study, Specialisation:	Structural Engineering, Structural Mechanics
Faculty:	TU Delft Faculty Of Civil Engineering and Geo sciences

Cover:	Picture modified by Isa Ritfeld
Style:	TU Delft Report Style, with modifications by Isa Ritfeld

TU Delft

Preface

This research is conducted to fulfill the requirements for the Master degree in Structural Engineering, with a specialization in Structural Mechanics, at the Technical University of Delft. The research is carried out from October 18th to August 23th, in cooperation with TNO, a Dutch applied scientific research organization, and Aronsohn, an engineering firm specializing in building design. It involves two high-rise buildings, the New Orleans and the Delftse Poort, both located in Rotterdam.

The research combines two of my greatest interests. The first interest is high-rise buildings, which I developed during my work experience at Aronsohn. There, I had the opportunity to work on building projects as a junior structural engineer. The complex modeling and calculations fascinated me. My other interest involves construction mechanics. Previously, I worked as a Teacher Assistant (TA) for the mechanics courses in the Bachelor's program of Civil Engineering. While the research focuses on the dynamics of buildings, a mechanical understanding, such as grasping the behavior of Finite Element (FE) models, is required to interpret the outcomes of this research. Moreover, as I had limited chances during my Master to see dynamics applied on such a scale. Therefore, this research provided a unique opportunity to explore the practical application of dynamics.

My experience with the research was highly positive. Through this research, I gained substantial knowledge, ranging from theoretical concepts to practical applications. I had the freedom to explore my own curiosities and discuss my findings with five great experts in the field. Communication is crucial, not only due to the involvement of multiple parties but also to enhance the research quality. Therefore, I would like to thank the supervisors Karel van Dalen, Eliz-Mari Lourens, Paul Lagendijk, Okke Bronkhorst, and Davide Moretti for guiding this thesis and allowing me to grow along the way. Moreover, I extend my gratitude to TNO and Aronsohn for providing me with weekly guidance, a workspace to conduct this research, and the necessary equipment. Lastly, I want to thank my family and friends for their interest and unwavering support during this time. I hope the reader will be as enthusiastic about reading this research as I was during the process.

Isa Sophie Ritfeld
Delft, August 2024

Summary

This research focused on applying vibration-based model updating to estimate the structural properties of high-rise buildings using a discrete Timoshenko beam model. Previous studies by Moretti et al. [1] and Taciroglu et al. [2] showed limitations in estimating the structural properties of buildings using model updating with a uniform Euler-Bernoulli beam model or a uniform Timoshenko beam model. Both models were not able to describe the third measured bending mode accurately after updating. Moreover, with the uniform Euler-Bernoulli beam model, when only the lowest two bending modes were used to fit the model, the estimates of the rotational stiffness of the foundation around the y-axis, $K_{r,y}$, showed significant uncertainty after model updating, with a Coefficient of Variation (CV) of 46.9%. Therefore, this research aimed to improve the accuracy of structural property estimations for high-rise buildings by using a more detailed model.

The model updating method applied in this research was an indirect vibration-based technique, which adjusted the input parameters of the chosen model to minimize the difference between the model output and the measured data. Both the model output and the measured data were in the shape of natural frequencies and mode shapes. The technique was applied to two high-rise buildings: the residential tower New Orleans, which is bending-dominant in behavior, and the office tower Delftse Poort, which is shear-dominant in behavior and exhibits irregular stiffness across its height. The discrete Timoshenko beam models used to approximate the dynamic behavior of these buildings were created using the Finite Element (FE) models of the buildings.

The research findings highlighted several key aspects essential for obtaining more accurate estimations of the structural properties of high-rise buildings. For a more accurate estimation of parameter $K_{r,y}$, it is of importance to incorporate shear deformations in the model and to account for irregular stiffness along the height. For the New Orleans, using the discrete Timoshenko beam model led to estimates of parameter $K_{r,y}$ with low uncertainty, indicated by a Coefficient of Variation of 6.91%. This was an improvement compared to the study by Moretti et al. [1], which showed a Coefficient of Variation of 46.9% using the uniform Euler-Bernoulli model.

Moreover, obtaining accurate values for the bending stiffness was crucial for achieving more precise estimations of the structural properties. It was challenging to determine the bending stiffness values using the FE models. These challenges posed a problem for model updating, as the initial structural property ratios were maintained for both high-rise buildings. Maintaining these ratios gives weight to the initial structural property values. These values must be correct. Otherwise, model updating is limited by the incorrect ratios and will not be able to accurately match the measured modal properties.

For the Delftse Poort, more pure bending modes were needed for better accuracy. The discrete Timoshenko beam model was unable to match the measured second bending mode in the y-direction, as it exhibited twisting, which the model could not represent due to its limitation to pure bending modes. Furthermore, since the discrete Timoshenko beam model requires a single displacement value per height but multiple sensors were used to measure displacements, the measurements had to be averaged. This averaging process may led to a mode shape that deviates from the actual behavior, resulting in an inaccurate representation of reality.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

Abbreviation	Definition
CMC	Crude Monte Carlo
DE	Diffrential Evolution
FEM	Finite Element model
FRF	Frequency Response Function
MAC	Modal Assurance Criterion
PSO	Partical Swarm Optimisation

Symbols

Symbol	Definition	Unit
A	Cross-sectional area	[m^2]
L	Length	[m]
ρ	Mass density	[kg/m^3]
M	Mass	[kg]
I	Second moment of Inertia	[m^4]
E	Elastic modulus	[N/m2]
EI	Bending stiffness	[Nm2]
v	Poisson ratio	[-]
G	Shear modulus	[N/m2]
k	Shear coefficient	[-]
K_r	Rotational spring stiffness	[Nm/rad]
K_t	Translational spring stiffness	[N/m]
G_s	Soil shear modulus	[N/m]
ρ_s	Soil mass density	[kg/m^3]
θ_s	Poisson ratio soil	[-]
e	Embedment depth	[m]
a_0	Dimensionless frequency	[-]
V_s	Shear wave velocities	[m/s]
f_n	Natural frequency	[Hz]
ϕ	Mode shape	[-]
CV	Coefficient of Variation	[-]
x	Direction main building axis	[-]
y	Direction orthogonal to x	[-]
z	Vertical building axis	[-]
σ	standard deviation	[-]
μ	mean	[-]

1 Introduction

In structural engineering, Finite Element (FE) models are used for simulations, design, and risk assessment. These models are solved using the Finite Element method, a numerical approach that divides a system into well-defined components known as elements. By subdividing a system into well-defined elements, whose behavior is completely understood, it becomes possible to estimate the behavior of such systems [3].

According to Zienkiewicz and Taylor [4], the FE method is considered the most appropriate tool for numerical modeling, as it can handle complex structural geometries, large assemblies of structural components, and various types of analysis. However, while FE models strive to accurately replicate the physics of their corresponding structures, discrepancies still exist between the measured and model output responses. This is due to the presence of measurement and modeling errors [3].

Measurement errors consider the inaccuracies of the measured data and arise from random noise in the measurements or from the measurement system setup. Modeling errors, on the other hand, consider the inaccuracies of the model prediction and arise from uncertainties in the geometry or boundaries, inaccurate simplifications or discretization, or uncertainty in the governing physical equations of the system [4, 5].

Due to awareness of modeling errors, efforts have been made to develop methods that improve models using measurements. This area is known as system identification and falls under control theory, a field dealing with the control of dynamical systems in engineering [6]. One technique within this area is vibration-based model updating, a technique that uses vibration measurements to enhance model accuracy. This technique includes several methods, one of which, called the indirect method, adjusts the model input parameters values to minimize the difference between the model output and the measured data. Both the model output and the measured data are in the shape of natural frequencies and mode shapes with this method.

In research, attempts have been made to use this indirect method of vibration-based model updating to estimate the structural properties of buildings. Moretti et al. [1] applied the technique to estimate the structural properties of the residential tower New Orleans in Rotterdam, utilizing a uniform Euler-Bernoulli beam model to approximate the dynamic behavior of the building. Similarly, Taciroglu et al. [2] applied the technique to estimate the dynamic stiffnesses of the soil foundation of the multi-storey building Millikan Library in Pasadena, using a uniform Timoshenko beam model to approximate its dynamic behavior. The structural properties were the input parameters of the models. Both studies used the lowest three measured bending modes to fit the model.

1.1. Research problem statement

While the study by Moretti et al. [1] showed that the uniform Euler-Bernoulli beam model accurately describes the lowest two measured bending modes after model updating, it also demonstrated that the chosen model fails to accurately describe the third measured bending mode after updating.

Moreover, when only the lowest two bending modes were used to fit the model, the estimates of the rotational stiffness of the foundation around the y-axis, $K_{r,y}$, showed large uncertainty after model updating, with a Coefficient of Variation (CV) of 46.9%, which was 6 to 7 times higher than that of other updating parameters. Similarly, for the Millikan Library building, the study by Taciroglu et al. [2] showed that the uniform Timoshenko beam model also failed to accurately describe the third measured bending mode after model updating.

According to Moretti et al. [1], the uniform Euler-Bernoulli beam model did not include all necessary aspects to accurately describe the third bending mode, limiting the accuracy of the results obtained. Additionally, they suggested that the large uncertainty in parameter $K_{r,y}$ may also be related to the model choice, as the chosen model assumed uniform stiffness across the height and does not account for the frequency dependency of the foundation stiffness. Similarly, Taciroglu et al. [2] suggested that the discrepancy between the estimated and third measured bending mode was due to the modeling choice.

A more detailed model could provide insights into the observed discrepancy in the third bending mode and offer explanations for the large uncertainty obtained in the estimations of parameter $K_{r,y}$. Therefore, this research investigates the effectiveness of vibration-based model updating in estimating the structural properties of high-rise buildings using a discrete Timoshenko beam model.

1.2. Research objectives

The research has two objectives:

- **Main objective:** a more accurate estimation of the structural properties of high-rise buildings using vibration-based model updating.
- **Secondary objective:** the application of vibration-based model updating with a discrete Timoshenko beam model.

1.3. Research questions

The main question the research aims to answer is:

How can the structural properties of high-rise buildings be more accurately estimated using vibration-based model updating?

The sub-questions that help answer the main question are:

- What is vibration-based model updating and which methods exist in literature?
- How is vibration-based model updating applied on buildings and what are the findings?
- What are the choices in the settings of the technique, and how do these choices influence the model updating results?
- What is a Timoshenko beam model and what are important aspects of the model?
- How can the Timoshenko beam model be used to approximate the dynamic behavior of high-rise buildings?
- How effective is vibration-based model updating in estimating the structural properties of high-rise buildings using a discrete Timoshenko beam model?

1.4. Significance research

The research provides valuable insights into the impact of a discrete Timoshenko beam model on the estimation of structural properties for high-rise buildings. Moreover, it expands the application of vibration-based model updating to high-rise buildings that are shear-dominant and have irregular stiffnesses across their height.

1.5. Scope

The research focuses on modeling changes within the superstructure of the beam model. Model updating is only conducted using bending modes. Torsional and mixed modes are outside the scope of this study. Nonlinear behavior in modal properties, such as the effects of vibration amplitude or building lifespan on the natural frequencies, is not considered.

1.6. Limitations

Due to the variability of high-rise building in type and behavior, the results of the research cannot be universally applied to all high-rise structures. The studies of the research are conducted on two specific in-situ high-rise buildings: the residential tower New Orleans, which is bending-dominant in behavior, and the office tower Delftse Poort, which is shear-dominant in behavior and has irregular stiffness across its height. Therefore, the results can only be applied to high-rise buildings with similar types and behaviors to those of the New Orleans and Delftse Poort

In addition, the research acknowledges the limitations of the discrete Timoshenko beam model, as it remains a simplified beam model and therefore may not fully capture the complex behavior of the high-rise buildings in question. While the model can accurately estimate the overall behavior of the building, small deviations from the actual behavior are expected. Therefore, the model updating results of the structural properties should be considered indicative rather than precise.

Furthermore, the research includes studies that investigate the influence of certain settings and uncertainties in model updating using a discrete Timoshenko beam model. These studies are limited to examining the effects of model choice, its parameters, the optimization algorithm, the number of measured modal properties, and measurement uncertainties on the results obtained. The optimization algorithms are kept at their default settings.

1.7. Organisation research

This report starts with a theoretical background, explaining what vibration-based model updating is and reviewing the methods available in literature. Chapter 3 discusses its application on buildings. Chapter 4 describes the methodology of the research, including a description of the model updating method applied and the discrete Timoshenko beam model used. Chapter 5 presents the studies conducted on the residential tower New Orleans, evaluating the effectiveness of model updating in estimating the structural properties of a bending-dominant building. Chapter 6 presents the studies conducted on the office tower Delftse Poort, assessing the effectiveness of model updating in estimating the structural properties of a shear-dominant building with irregular stiffness across its height. The research concludes with a discussion and conclusion.

2

Theoretical background

This chapter explains what vibration-based model updating is and outlines the methods available in the literature. The chapter begins with Section 2.1, which explains the background of vibration-based model updating. Next, Section 2.2 explains the key concepts and theories underlying the technique. Section 2.3 explains the type of optimization algorithms which can be applied with the technique. Finally, Section 2.4 presents the methods of vibration-based model known in literature.

2.1. Background

In structural engineering, Finite Element (FE) models are used for simulations, design, and risk assessments. The models are solved using the Finite Element method, a numerical technique that divides a system into a finite number of well-defined components called elements. These elements create a FE mesh, as shown in Figure 2.1. By subdividing a system into well-defined elements, whose behavior is completely understood, it becomes possible to estimate the behavior of such systems [3].

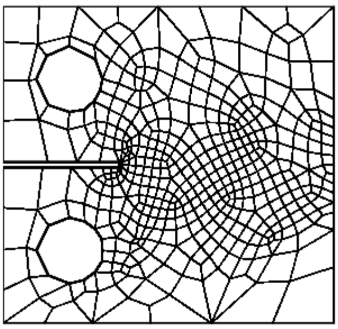

Figure 2.1: A typical two-dimensional finite element mesh [3].

According to Zienkiewicz and Taylor [4], the FE method is considered the most appropriate tool for numerical modeling, as it is capable of handling complex structural geometries, large assemblies of structural components, and different types of analysis. However, while FE models attempt to exactly replicate the physics of their corresponding structures, discrepancies still exist between the measured and model output responses. This is due to the presence of measurement and model errors.

Measurement errors consider the inaccuracies of the measured data and are encountered as random or systematic errors. Random errors arise from random noise in the measurements and are by nature unpredictable. Systematic errors arise from the measurement system setup and are encountered as [4]:

- Mounting errors, where incorrect mounting of the equipment does not allow accurate modeling of the system.

- Mass or stiffening errors, where attachment of equipment to the system causes mass loading and local stiffening.

The processing of data may also produce errors. Random and systematic errors can never be completely eliminated. Therefore, ensuring high-quality measurements is essential to minimize these errors. According to Friswell, there is no substitute for high-quality measurements [7].

Modeling errors consider the inaccuracies of the model prediction and are encountered as [5]:

- Model structure errors, which occur when there is uncertainty in the governing physical equations of the system.
- Model parameter errors, which involve uncertainties in properties and geometry, inaccurate application of boundary conditions, or inaccurate simplification of the model.
- Model order errors, which arise due to the discretization of complex systems, resulting in poor replication of the system.

Due to the awareness of modeling errors, efforts have been made to develop methods that improve models using measurements. This area is known as system identification and falls under control theory, a field dealing with the control of dynamical systems in engineering [6]. Vibration-based model updating falls within this area.

According to Natke, model updating can be seen as an indirect type of system identification, where 'indirect' indicates the use of a reference model to represent the system [8]. According to Friswell, model updating can be seen as a parameter estimation problem, an area within system identification where vibration measurements are used to correct the parameters of the reference model [7]. Both definitions indicate the aim of the technique to improve the accuracy of models using vibration measurents.

The technique includes several methods for updating models. However, before these methods can be explained, the key concepts and theories need to be explained, as these form the basis of the different methods known. Therefore, the key concepts and theories are explained in the next Section (Section 2.2). The methods are explained in Section 2.4.

2.2. Key concepts and theories

Within the area of model updating, several key concepts and theories exist that form the basis of model updating. These key concepts and theories are in detail explained in the sections below.

2.2.1. Parameters

Parameters refer to the variables or factors within a structural model that influence its behavior or output. They are the primary inputs of the system and may include physical parameters, mass matrices, or stiffness matrices [9].

Physical parameters are intrinsic properties of the system, such as material properties (density, elasticity), geometric properties (dimensions, shapes), or boundary conditions. Stiffness and mass matrices define the relations between the physical parameters and the model output [9].

Research in this field has primarily focused on updating the physical parameters of the model [9].

2.2.2. Solution space

A solution space refers to an area containing all possible combinations or configurations of parameters that can be considered when searching for an optimal solution to a problem. It represents the range of feasible solutions, called candidate solutions, within a given problem domain, as illustrated in Figure 2.2.

The dimensions depend on the number of updated parameters. For example, with two variables, the search space is two-dimensional (2D). The solution space is unimodal if only one single solution exists and multimodal if it contains multiple local solutions, with one being the global solution of the given problem [10].

Real-world problems often involve complex, multidimensional multimodal solution spaces [10].

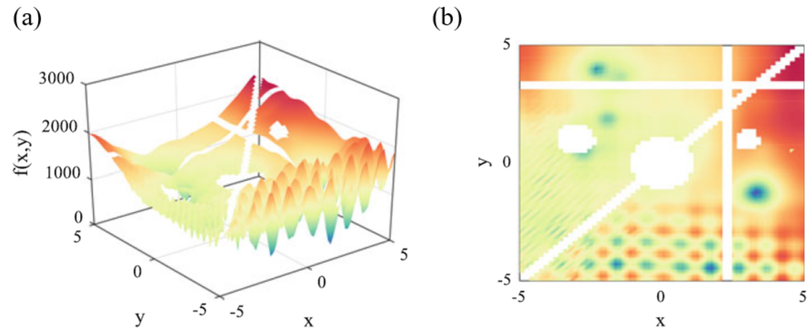

Figure 2.2: The solution space (a) of the variables x and y with several constraints. The top view is indicated with (b). The axis $f(x, y)$ indicates the fitting of the solutions. The constraints are represented with white shading within the solution space, indicating infeasible solutions [10].

2.2.3. Parametrization

Parametrization refers to the process of reducing the set of updating parameters in order to help ill-posedness or ill-conditioning of the problem. Model update problems are often ill-posed, indicating the absence of a unique solution. Even when the problem is well-posed, it can still be ill-conditioned, as minor fluctuations in measured data or the initial model state can significantly influence the updating outcomes [7, 9]. Therefore, parametrization is the key to success in model updating. According to Friswell and Mottershead, three essential conditions should be satisfied to achieve successful parameterization [5]:

- Limit the number of parameters to avoid ill-posedness
- Choose parameters that address the model uncertainties
- Ensure that the model outputs are sensitive to the chosen parameters

2.2.4. Constraints

There are two types of constraints. Explicit constraints refer to the constraints directly specified in the problem formulation, indicating the limits of the system (infeasible solutions). They are considered secondary inputs of the system and depicted as white areas in Figure 2.2. Penalty functions can be applied to handle these constraints during model updating. An example of a penalty function is illustrated in Figure 2.3. To ensure that the constraints are not violated, a high penalty (indicated in dark red) is assigned to these constraints. This ensures that they are not considered as solutions [10].

Implicit constraints, in contrast, are not explicitly defined in the problem formulation. They refer to the constraints introduced during the optimization process to help the ill-posedness or ill-conditioning of the problem (regularization) [7, 9].

2.2.5. Regularization

Regularization refers to the process of imposing additional constraints (implicit constraints) to a problem in order to help the ill-posedness of the problem. Model update problems are often ill-posed, indicating the absence of a unique solution. Even when the problem is well-posed, it can still be ill-conditioned, as minor fluctuations in measured data or the initial model state can significantly influence the updating outcomes. Regularization modifies the problem to make it better conditioned.

It involves adding penalty terms to the objective function being optimized. Objective functions are explained in Section 2.2.8. These penalty terms ensure that parameters are adjusted minimally during optimization, guiding the process towards simpler and more stable solutions [7, 9].

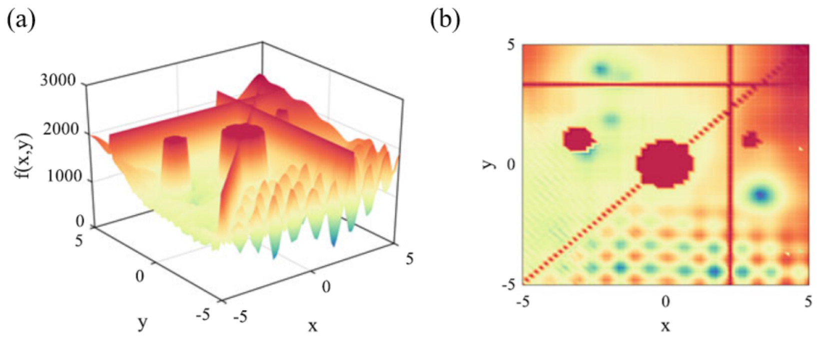

Figure 2.3: The application of the barrier penalty function on the solution space (a). The top view is indicated with (b). The axis $f(x, y)$ indicates the fitting of the solutions.

2.2.6. Optimization Algorithm

Optimization algorithms refer to the methods or procedures used to search for the global solution within the solution space of a given problem. The different types of algorithms are explained in Section 2.3. They initiate the process by generating a random set of candidate solutions (feasible solutions) within the solution space. They refine these candidate solutions to optimize or improve a certain objective function until an optimal solution is found. The manner in which these solutions are refined depends on the type of algorithm applied. [10].

2.2.7. Exploration and exploitation

Exploration and exploitation refer to different strategies used by algorithms to refine the candidate solutions within the solution space. With exploration, the candidate solutions are adjusted to explore different regions and to identify promising areas of the solution space. With exploitation, the best solution found so far is identified within the candidate solutions [10].

2.2.8. Objective function

An objective function refers to a mathematical function that quantifies the difference or discrepancy between the model output and the observed data. They evaluate the fitting of the candidate solutions. The aim is to minimize the objective function. The shape of the objective function depends on the method applied. The methods are explained in Section 2.4.

2.3. Optimization algorithms

This section describes the different types of algorithms which can be applied with the technique. The algorithms are divided into two categories: individual-based algorithms and population-based algorithms. These categories are explained in Sections 2.3.1 and 2.3.2, including a list of the limitations and strengths of each group.

2.3.1. Individual-based algorithms

Individual-based algorithms update the model by considering only one candidate solution, which is adjusted and evaluated until the objective function is minimized. The limitations and strengths of these group of algorithm are listed in Table 2.1 [10].

A strength of individual-based algorithms is their minimal need for function evaluations, as only one solution is considered. This can be advantageous in scenarios where function evaluations are computationally expensive. However, a limitation is their susceptibility to premature convergence. Real-world problems often involve complex, multidimensional solution spaces. Consequently, the algorithm may quickly converge to a local optimum, resulting in suboptimal performance on challenging optimization problems [10].

Strengths	Limitations
+ Minimal need for function evaluations	- Premature convergence

Table 2.1: The strengths and limits of Individual-based algorithms.

2.3.2. Population-based algorithms

Population-based algorithms update the model by considering a set of candidate solutions, which are evaluated and adjusted until the objective function is minimized. They are stochastic algorithms and rely on probabilistic rules. They are categorized into two groups: evolution-based algorithms and swarm-based algorithms [10].

Evolutionary algorithms mimic natural processes (crossover and mutations) to solve the optimization problem. The processes are visualized in Figure 2.4. Through crossovers, candidate solutions are combined to exploit the search space. Through mutations, candidate solutions are altered to explore the solution space [10].

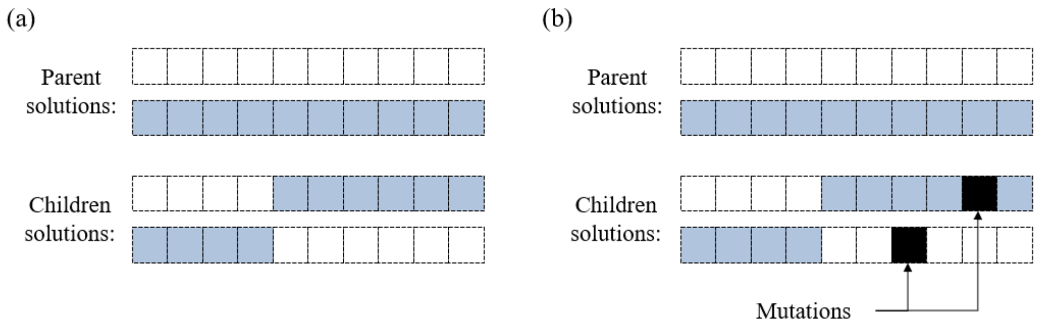

Figure 2.4: The processes of nature mimicked with evolutionary-based algorithms: (a) illustrates the crossover process, used to exploit the search. (b) Depicts the mutation process, used to explore the solution space.

Swarm algorithms use position vectors to solve optimization problems. Position vectors denote the positions of the candidate solutions within the search space. During the updating process, the positions are updated according to specific movement rules dictating direction and speed.

Initially, the solutions move quickly in diverse directions to explore the search space. As the algorithm progresses through iterations, the magnitude of movement decreases to fine-tune the search and exploit the best positions discovered during the exploration phase [10]. The limitations and strengths are listed in Table 2.2.

A strength of population-based algorithms is their lower probability of becoming trapped in local optima: if one solution gets stuck in a local optimum, other solutions can assist in avoiding it in other iterations. However, a limitation is the requirement for more function evaluations, which increases the computational expense of these algorithms [10].

Population-based algorithms use both exploration and exploitation to refine candidate solutions. However, using these algorithms presents a challenge, as it requires a balance between exploration and exploitation. These two aspects are in conflict: a more exploratory behavior may lead to finding solutions of poor quality, as the algorithm never has a chance to improve solution accuracy. On the other hand, a more exploitative behavior risks trapping the algorithm in local minima because it does not sufficiently adapt solutions [10].

Strengths	Limits
+ Lower probability of entrapment in local optima	- More function evaluations needed - The balance between exploration and exploitation

Table 2.2: The strengths and limits of population-based algorithms.

2.4. Methodologies

This section describes the existing methods for model updating. The methods are divided into three categories: the direct methods, the indirect methods, and the methods using frequency domain data. For each group of methods, the limitations and strengths are listed. The details are given in the Sections below.

2.4.1. The direct methods using modal data

With direct methods, the models themselves (the stiffness and mass matrices) are modified to update the models. The methods falling into this category are listed in Table 2.3. The limitations and strengths of these methods are listed in Table 2.4.

Category	Methods	References
Direct methods using modal data	Reference basis methods	Baruch and Berman (1978)
	Matrix mix approach	Thoren (1972) and Ross (1971)
	Eigenstructure assignment method	Minnas and Inman (1988)
	Pole placement method	Minnas and Inman (1988)
	Inverse eigenvalue methods	Glaswell (1986)

Table 2.3: An overview of the direct methods [5].

A strength of these methods is their ability to exactly reproduce the measured modal data. They do not need iterative refinement or extensive batch processes, which makes them computationally cheaper. A limitation, however, is the additional requirement for accurate modeling and high-quality measurements. It is unlikely that the measured and numerical data will be exactly equal due to the presence of measurement and modeling errors [5, 11].

Another limitation is their physical meaning. The updated matrices are generally fully populated, while the initial matrices contain only non-zero values on their diagonal. Furthermore, the positive definiteness of the updated mass and stiffness matrices, as well as the connectivity of the nodes, is not guaranteed [5].

Morover, because lower frequency modes are measured and higher frequency modes contribute most to the stiffness matrices, interpreting the results becomes complicated [5].

Strengths	Limitations
+ Able to exactly reproduce the measured modal data + Computationally cheaper	- Accurate modeling and high-quality measurements needed - Physical meaning updated matrices - Interpretation updated matrices

Table 2.4: The strengths and limits of the direct methods.

2.4.2. Indirect methods using modal data

With indirect methods, the input parameter values are adjusted to update the model. These methods are executed by an algorithm and involve the use of an objective function, denoted as J [1]:

$$J = w_f \sum_{i=1}^{N} \frac{|f_{n,i} - \hat{f}_{n,i}|}{\hat{f}_{n,i}} + w_\phi \sum_{i=1}^{N} (1 - MAC(\phi_i, \hat{\phi}_i)) \tag{2.1}$$

where index i denotes the mode considered, $f_{n,i}$ and ϕ_i denotes the estimated modal properties, and the $\hat{f}_{n,i}$ and $\hat{\phi}_i$ denotes the measured modal properties. The first term of the objective function J indicates the mismatch between the obtained and measured frequencies. The second term of the objective function J indicates the mismatch between the obtained and measured mode shapes. To quantify the mode shape mismatch, the Modal Assurance Criterion (MAC) is used [1]:

$$MAC(\phi_i, \hat{\phi}_i) = \frac{|\phi_i \cdot \hat{\phi}_i|^2}{(\phi_i \hat{\phi}_i)(\phi_i \cdot \hat{\phi}_i)} \tag{2.2}$$

The MAC returns a value between 0 and 1, indicating no or complete similarity between the two mode shapes. The weight factor w_f or w_ϕ determine the contributions of the terms in the objective function [1] [5]. The methods falling into this category are listed in Table 2.5. The limitations and strengths of these methods are listed in Table 2.6.

Category	**Methods**	**References**
Iterative methods	Penalty functions methods	Friswell and Mottershead
using modal data	Minimum Variance methods	Collins et al. (1972)

Table 2.5: An overview of the iterative methods of the model updating technique [5].

The strength of these methods lies in the wide choice of parameters and the ability to weigh the terms of the objective function, which gives them power and versatility. However, it does require complete understanding which weigh factors to use [5].

Another limitation is the often ill-posed nature of the problem, indicating the potential absence of a unique solution. Additionally, long computational times can be expected due to the need to evaluate the model at every iteration [5]. Convergence problems may also arise [7].

Strengths	**Limitations**
+ Wide choice of parameters	- Understanding which weigh factors to use
+ Weighing the terms objective function	- Ill-posed nature of the problem
	- Long computational times
	- Convergence problems

Table 2.6: The strengths and limits of the indirect methods.

Additionally, it is essential for these methods that the model and measured data correspond to the same mode, which can pose a significant challenge. Simply arranging the natural frequencies in ascending order of magnitude may not suffice, especially when two modes are closely located in the frequency domain. Moreover, obtaining accurately measured modes can be problematic, and certain modes may not be excited during the experiment, resulting in no corresponding measured mode for the analytical modes [5].

2.4.3. Methods using Frequency Domain Data (FDD)

The last group involves methods that use an objective function directly based on the Frequency Response Function (FRF) data to update the model. The methods falling within this category are listed in table 2.7. Both methods are based on the equation of motion in the frequency domain for viscous damping:

$$[-\omega^2 M + i\omega C + K]x(\omega) = f(\omega) \tag{2.3}$$

The equation error approach minimizes the error in the equation of motion, given as:

$$\epsilon_{ee} = f(\omega) - [-\omega^2 M + i\omega C + K]x(\omega) \tag{2.4}$$

Where $x(\omega)$ an $f(\omega)$ are measured quantities. The output error approach minimizes the output error, defined as the difference between the measured and the estimated response:

$$\epsilon_{oe} = [-\omega^2 M + i\omega C + K]^{-1} f(\omega) - x(\omega) \tag{2.5}$$

Because the Frequency Response Functions (FRFs) are directly used, the force and displacement are not available individually. Therefore, with both approaches, the force spectrum is assumed to be white noise, and the displacement is replaced by the FRFs data. The limitations and strengths are listed in Table 2.8.

Category	Methods	References
Methods using FDD	Equation error approach Output error approach	Friswell and Penny (1992)

Table 2.7: An overview of the iterative methods of the model update technique [5].

Both methods eliminate the need for extracting natural frequencies and mode shapes, which can be challenging for structures with closely spaced modes or high modal density. Although Frequency Response Functions (FRFs) contain more data than the modal model, it should however not be assumed that they provide a proportionally increased amount of information [5].

However, a significant limitation is that damping must be included in the finite element model. Damping is important for achieving good correspondence between the measured and predicted FRFs, however, it is difficult to model accurately. Methods that use modal data do not face this issue, as they can rely on undamped models. This is possible because natural frequencies and damping ratios can be separated [5].

Strengths	Limitations
+ No need for extraction modal properties	- The inclusion of damping

Table 2.8: The strengths and limits of the methods using FDD.

2.5. Conclusion

This chapter provided an in-depth explanation of vibration-based model updating, a technique that improves models using vibration measurements. This technique falls within the domain of system identification and falls under the control theory, a field dealing with the control dynamic systems in engineering. The technique uses a model to represent the system and includes several methods:

- **Direct methods** modify the model themselves (the stiffness and mass matrices) to align the model output with the measured data. These methods can exactly reproduce the measured modal data. However, they require accurate modeling and high-quality measurements. Another limitation is the physical meaning, as the updated matrices are generally fully populated, while the initial matrices typically contain non-zero values only on their diagonals.
- **Indirect methods** adjust the input parameter values to minimize the difference between the model output and the measured data. Both the model output and the measured data are in the shape of natural frequencies and mode shapes. The strength of these methods lies in the wide choice of updating parameters, giving them power and versatility. However, the problems are often ill-posed, meaning there may be no unique solution. Additionally, long computational times can be expected due to the need to evaluate the model every time a change in the input parameter values is made.

- **Methods based on Frequency Domain Data** utilize an objective function directly based on the FRF data to update the model. These methods eliminate the need to extract modal properties, which can be challenging for structures with closely spaced modes or high modal density. However, a significant limitation is the necessity of including damping in the finite element model. Damping is crucial for achieving good correspondence between the measured and predicted FRFs, but it is difficult to model accurately.

While each method for vibration-based model updating offers unique advantages, they also come with specific challenges and limitations. How vibration-based model updating is applied on buildings is show in the Chapter 3.

3

Literature review

This chapter shows how vibration-based model updating is applied on building. It describes two applications, which both use the technique to estimate the structural properties. The first application is a study conducted by Moretti et al. [1] on the residential tower New Orleans in Rotterdam, explained in Section 3.1. The second application is a study on the Millikan Library building in Pasadena conducted by Taciroglu et al. [2], explained in Section 3.2. Their findings are highlighted at the end.

3.1. The residential tower New Orleans in Rotterdam

The first application involves a study on the residential tower New Orleans in Rotterdam, the Netherlands, conducted by Moretti et al. [1]. The high-rise building is shown in Figure 3.1.

Figure 3.1: A picture of the residential tower New Orleans.

High-rise buildings are sensitive to wind-induced vibrations, making these vibrations important for the design of the serviceability limit state (SLS) and the ultimate limit state (ULS).
The main parameters of these vibrations are the eigenfrequencies and damping ratios. Accurately estimating these parameters during the design phase is difficult, as they depend on factors such as building mass, building stiffness and foundation stiffness. Therefore, to obtain a better prediction of the dynamic behavior, a better prediction of these structural properties is needed.

To obtain this, it is important to know the actual values of these structural properties and how they relate to the design values. Therefore, to obtain this information, an indirect method of vibration-based model updating is used, explained in Section 2.4.2. The weight factors of the objective function (Equation 2.1) are set to 1.

The model applied to approximate the dynamic behavior of the New Orleans is a beam model and is shown in Figure 3.2. The superstructure is modeled as a uniform Euler-Bernoulli beam with bending stiffnesses EI_x and EI_y, and a mass per unit length ρA. The foundation is modeled as a combination of rotational ($k_{r,x}$ and $k_{r,y}$) and translational springs ($k_{t,x}$ and $k_{t,y}$). Due to the choice of the model, only bending modes can be considered in the updating process.

The initial input parameter values are given in Table 3.1. The measured modal data applied to fit the model are listed in Table 3.2, obtained by analyzing vibration measurements on the 15th, 34th, and 44th floors. The measurement setup used to obtain the vibration measurements is shown in Figure 3.3.

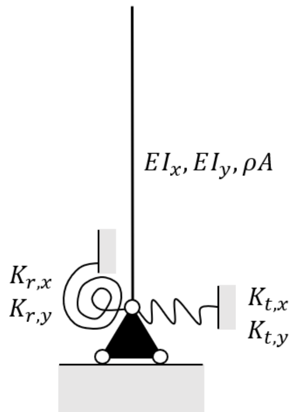

Figure 3.2: The uniform Euler-Bernoulli beam model (2D) used to approximate the dynamic behavior of the high-rise building.

Properties [unit]	
EI_x [Nm^2]	$3.06e13$
EI_y [Nm^2]	$2.43e13$
$K_{r,x}$ [Nm/rad]	$1.88e12$
$K_{r,y}$ [Nm/rad]	$1.88e12$
$K_{t,x}$ [N/m]	∞
$K_{t,y}$ [N/m]	∞
M [kg]	$6.50e7$
L [m]	155
A [m^2]	841
ρ [kg/m^3]	500

Table 3.1: The design values assigned as initial input parameter values for the Euler-Bernoulli beam model [1].

Mode	**1**	**2**	**3**	**4**	**5**
Dominant direction	X	Y	Y	X	Y
Natural frequency [Hz]	0.282	0.291	1.332	1.527	2.771
Modal displacement					
Height 1 (51.4 m)	−0.30	−0.29	−0.90	0.89	−0.73
Height 2 (114.6 m)	−0.71	−0.78	0.19	−0.23	0.83
Height 3 (147.9 m)	−1.00	1.00	1.00	−1.00	−1.00

Table 3.2: The identified pure bending modes in dominant x- or y- direction [1, 12].

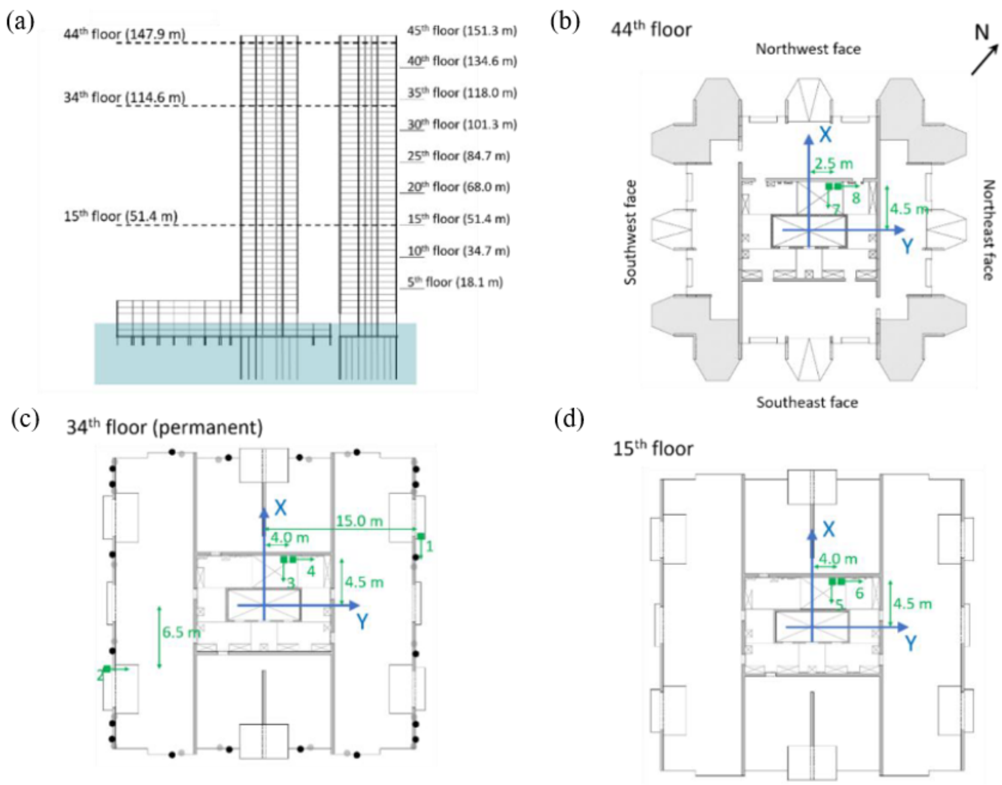

Figure 3.3: (a) Picture and (b, c, d and e) drawings indicating the acceleration sensors positions (green squares) on the 15th floor, the 34th floor, and the 44th floor. The green arrows indicate the directions of the acceleration sensors [1].

Two scenarios are considered:

- Case 1: model updating with 2 bending modes in x- and y-direction.
- Case 2: model updating with 2 bending modes in x-direction and 3 bending modes in y-direction.

In both cases, the parameters EI_x, EI_y, Kr_x, Kr_y, Kt_x, Kt_y, and ρ are updated. The findings are given in Tables 3.3 and 3.4.

According to Table 3.3, both Case 1 and Case 2 show alignment with the measured modal data after updating. However, the accuracy ($\pm\sigma$) of Case 2 is lower.

The Modal Assurance Criterion (MAC) in Case 2 indicates that the third bending mode in y-direction is not properly matched. This suggests that the chosen model is only capable of accurately describing the lower two bending modes in both directions. According to Moretti et al. [1], with higher bending modes, other modeling aspects become important, which are not included in the chosen model.

Mode	**Design [Hz]**	**Measured [Hz]**	**Case 1 ($\pm\sigma$) [Hz]**	**Case 2 ($\pm\sigma$) [Hz]**	**MAC Case 1 %**	**MAC Case 2 %**
1 (x)	0.166	0.282	0.282 (± 5e-5)	0.286 (± 0.03)	99.84	97.58
2 (y)	0.153	0.291	0.291 (± 5e-5)	0.253 (± 0.02)	99.73	99.22
4 (y)	0.989	1.322	1.332 (± 5e-5)	1.333 (± 0.04)	99.90	98.47
5 (x)	1.089	1.527	1.516 (± 5e-5)	1.548 (± 0.08)	99.70	95.45
7 (y)	2.823	2.771	-	**3.111** (± 0.56)	-	**78.11**

Table 3.3: The modal properties of the residential tower New Orleans in terms of natural frequencies and MAC [1].

Looking at the results of the structural properties of Case 1 (Table 3.4), it can be observed that, except for ρ, the design values of the structural properties fall outside the value ranges obtained with updating.

The values obtained provide a good basis for making design choices that can lead to better prediction of the dynamic behavior.

However, this cannot be said for the rotational stiffness K_{ry}. A large uncertainty (46.9%) can be seen in the obtained results of the rotational stiffness K_{ry}, which is 6 to 7 times higher than that of other updating parameters. According to Moretti et al. [1], this may be also related to the choice of model, as the current model assumes uniform stiffness over the whole height and does not consider the frequency dependence of the foundation stiffness. A more detailed model could provide insight into the mismatch of the third bending mode and the observed uncertainty in the estimate for K_{ry}.

Property [unit]	**Design**	**Median**	**Lower bound** (±%)	**Upper bound** (±%)	**CV** (%)
EI_x [Nm^2]	3.06e13	9.70e13	8.62e13 (-11.1 %)	10.8e13 (+11.3 %)	7.0 %
$K_{r,x}$ [Nm/rad]	1.88e12	3.07e12	2.76e12 (-10.1 %)	3.33e13 (+8.6%)	5.8 %
$K_{t,x}$ [N/m]	∞	2.85e9	2.85e9 (-9.5 %)	3.07e13 (+7.7 %)	5.4 %
EI_y [Nm^2]	2.43e13	6.51e13	5.78e13 (-11.2 %)	7.50e13 (+15.2 %)	7.9 %
$K_{r,y}$ [Nm/rad]	1.88e12	1.11e13	0.64e13 (-42.6 %)	2.29e13 (+105.8 %)	**46.9** %
$K_{t,y}$ [N/m]	∞	2.20e9	1.98e9 (-10.3 %)	2.40e13 (+9.0 %)	5.9 %
ρ [kg/m^3]	500	468.6	432.0 (-7.8 %)	502.9 (+7.3 %)	4.7 %

Table 3.4: The structural properties results of the model update procedure of Case 1. For each property, the median value, the 90% confidence intervals and the coefficient of variation are computed. The design values are given for comparison [1].

3.2. The Millikan Library building in Pasadena

The second application is a study on the Millikan Library building at the California Institute of Technology in Pasadena, conducted by Taciroglu et al. [2]. The multi-storey building is show in Figure 3.4. This study presents a new approach to estimate the dynamic stiffnesses of soil foundation systems using model updating.

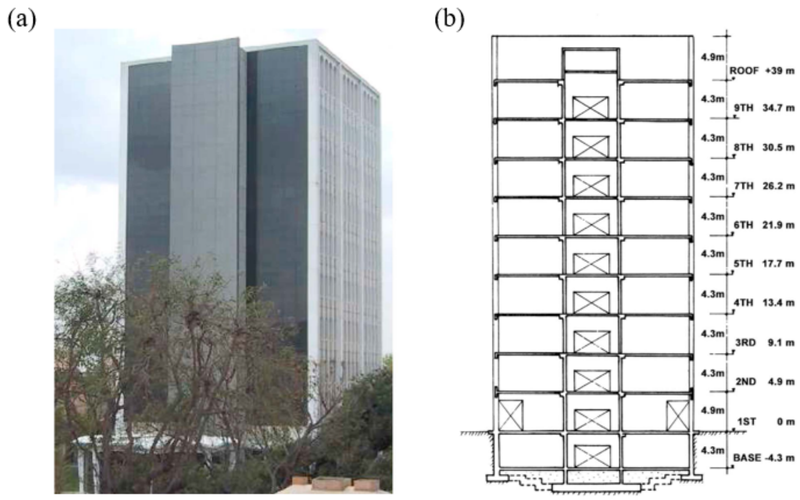

Figure 3.4: A picture (a) and a schematic (b) of the Millikan Library building [2].

The model chosen to approximate the dynamic behavior of the Millikan Library is a beam model and is shown in Figure 3.5. The superstructure is modeled as a uniform Timoshenko beam with bending stiffness EI, shear stiffness kGA and a mass per unit length ρA. The foundation is modeled as a combination of rotational ($k_{r,x}$ and $k_{r,y}$) and translational springs ($k_{t,x}$ and $k_{t,y}$) and includes frequency dependency. The model takes both bending and shear deformations into account.

Dimensionless parameters (equations 3.1 and 3.2) are defined through the relation of the input parameters:

$$b^2 = \frac{\rho A \omega^2 L^4}{EI}, s^2 = \frac{EI}{kGAL^2} \tag{3.1}$$

$$k_t(\omega_i) = \frac{K_t(\omega_i)}{kGA/L}, k_r(\omega_i) = \frac{K_r(\omega_i)}{EI/L} \tag{3.2}$$

where ω_i indicates the frequency dependency of the foundation. The initial input parameter values are given in table 3.5. The measured natural frequencies, obtained by analyzing vibration measurements from the 2002 Yorba Linda earthquake [13], are listed in Table 3.5.

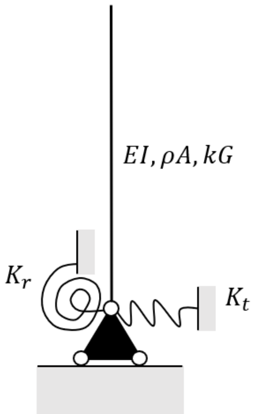

Figure 3.5: The uniform Timoshenko beam model (1D) use to approximate the dynamic behavior of the high-rise building.

Properties [unit]	
EI [Nm^2]	17.75e3
M [kg]	11.95e6
L [m]	43.28
A [m^2]	483

Table 3.5: The structural properties of the Millikan Library building [2].

Mode	**1**	**2**	**3**
Natural frequency [Hz]	1.68	6.64	12.48

Table 3.6: The natural frequencies identified from the 2002 Yorba Linda earthquake data [13]

Two scenarios are considered:

- Scenario I: the measurements of the rocking response of the foundation are considered.
- Scenario II: the measurements of the rocking response of the foundation are not considered.

When the measurements of the rocking response are considered (scenario I), an iterative method of the model updating is applied, which is explained in Section 2.4.2. The dimensionless parameters b, s, $k_r(\omega_1)$ and $k_t(\omega_1)$ are chosen as the updating parameters. Only the lowest measured bending mode is used to fit the model. The weight factors of the objective function indicated with Equation 2.1 are set to 1.

After updating the dimensionless parameters, the physical values of the dynamic stiffness of the soil foundation system are obtained using the Equation 3.3:

$$K_t(\omega_1) = \frac{K_t M}{b^2 s^2}, K_r(\omega_1) = \frac{K_t M L^2}{b^2} \tag{3.3}$$

When the measurements of the rocking response are not considered (scenario II), equations 3.4 till 3.9 are used to approximate the stiffness of a rigid rectangular foundation:

$$K_t(\omega_i) = K_t^s k_t^d(\omega_i) \tag{3.4}$$

$$K_t^s(\omega_i) = \frac{G_s D}{2(2-\theta_s)}[6.8(B/D)^{0.65} + 2.4 + 0.8(B/D-1)][1 + (0.33 + \frac{1.34}{1+B/D})(\frac{2e}{D})^{0.8}] \tag{3.5}$$

$$k_t^d(\omega_i) = 1 \tag{3.6}$$

$$k_r(\omega_i) = K_r^s k_r^d(\omega_i) \tag{3.7}$$

$$K_r^s = \frac{G_s D^3}{8(1-\theta_s}[3.2(B/D) + 0.8][1 + (\frac{e}{D} + \frac{1.6}{0.35 + B/D})(\frac{2e}{D})^2] \tag{3.8}$$

$$k_r(\omega_i) = 1 - \frac{d a_o(\omega_i)^2}{b + a_o(\omega_i)^2} \tag{3.9}$$

where k_t^d and k_r^d are dimensionless parameter representing the frequency dependency, and K_t^s and K_r^s are the static stiffnesses of the foundation. The parameter values are listed in Table 3.7. Functions of the dimension are listed in Table 3.8.

Property [unit]	**Value**
G_s [N/m]	2.68e8
ρ_s [kg/m^3]	2.7e3
θ_s [-]	0.3
B [m]	23
D [m]	21
e [m]	4

Table 3.7: The soil and foundation properties of the Millikan library building. G_s is the soil shear modulus, ρ_s is the soil mass density and θ_s is the Poisson ratio of the soil. B and D represent the dimension foundation and e denotes the embedment depth of the foundation.

Property [unit]	**Function**
a$_o$ [-]	$\frac{\omega_i D}{2V_s}$
V$_s$ [m/s]	$\sqrt{\frac{G_s}{\rho_s}}$

Table 3.8: The dimensionless frequency a_o and the shear wave velocity of the soil Vs.

The findings of the study are given in Table 3.9 and 3.10. To compare the obtained values with a previous study conducted by Ghahari et al. [13], where the FE model of the Millikan Library is directly updated using the same measured modal data, the sway and rocking stiffness values are normalized by Ga and Ga^3 respectively, where $G_s = 2.68e8\ N/m$ denotes the soil shear modulus and $a = 13.7\ m$ the reference foundation length.

According to Table 3.9, the stiffness values obtained using the presented methods are close to the identified values from the study conducted by Ghahari et al. [13].

Study	**Sway [-]**	**Rocking [-]**
Ghahari et al. [13]	6.2	3.9
Scenario I	6.7	4.2
Scenario II	6.7	4.1

Table 3.9: Normalized sway and rocking soil-foundation stiffnesses.

According to Table 3.10, the first two natural frequencies are close to the identified values. However, the third natural frequency shows a significant difference from the identified value.

Mode	Measured [13] [Hz]	FE model [13] [Hz]	Scenario I [Hz]
1	1.68	1.69	1.68
2	6.64	6.64	6.67
3	12.48	12.53	14.05

Table 3.10: The obtained and identified natural frequencies of the Millikan Library building.

To determine if this discrepancy is due to the choice of foundation, also a scenario with a fixed-base was considered. The results obtained with a fixed-base are given in Table 3.11.

The third natural frequency from the Timoshenko beam model with a fixed base show a 12% difference compared to the FE model. This suggests that the error is not solely due to the chosen foundation. A more detailed model is needed to predict the higher mode responses accurately.

Mode	Measured [13] [Hz]	FE model [13] [Hz]	Scenario I [Hz]
1	2.05	2.07	2.07
2	-	7.51	7.56
3	-	13.96	15.67

Table 3.11: The obtained and identified natural frequencies of the Millikan Library building using a fixed base.

3.3. Conclusion

This chapter shows how vibration-based model updating is applied to estimate the structural properties of buildings with two applications. The first application is a study on the residential tower New Orleans by Moretti et al. [1], which used an uniform Euler-Bernoulli beam model to approximate the dynamic behavior of the residential tower New Orleans. The second application is a study on the Millikan Library building in Pasadena by Taciroglu et al. [2], which used a uniform Timoshenko beam model to approximate its dynamic behavior.

The study of Moretti et al. [1] showed that while the uniform Euler-Bernoulli beam model is able to describe the lowest two bending modes after model updating, it fails to accurately describe the third measured bending mode after updating. Moreover, when only the lowest two bending modes were used to fit the model, the estimates of the rotational stiffness of the foundation in the y-direction, $K_{r,y}$ showed a large uncertainty after model updating, with a coefficient of variation of 46.9 %, which is 6 to 7 times higher than that of other updating parameters. Similary, the study of Taciroglu et al. [2] showed that the uniform Timoshenko beam after model updating failed to accurately describe the third measured bending mode.

According to Moretti et al. [1], the uniform Euler-Bernoulli beam model did not include all necessary aspects to accurately describe the third bending mode, limiting the accuracy of the results obtained. Additionally, they suggested that the large uncertainty in parameter $K_{r,y}$ may also be related to the model choice, as the chosen model assumed uniform stiffness across the height and does not account for the frequency dependency of the foundation stiffness. Similarly, Taciroglu et al. [2] suggested that the discrepancy between the estimated and third measured bending mode was due to the modeling choice.

Applying a more detailed model with vibration-based model updating could provide insights into the observed discrepancy in the third bending mode and offer explanations for the significant uncertainty encountered in the estimations of parameter $K_{r,y}$. Therefore, this research investigates the effectiveness of vibration-based model updating in estimating the structural properties of high-rise buildings using a discrete Timoshenko beam model. The methodology of the research is explained in Chapter 4.

4
Methodology

This chapter explains the research methodology, including a detailed description of the model updating method applied and the discrete Timoshenko beam model used. Section 4.1 discusses the vibration-based model updating method applied. Section 4.2 describes the discrete Timoshenko model.

4.1. The model updating method

For this research, an indirect method of vibration-based model updating is chosen, the same method applied by Moretti et al. [1] and Taciroglu et al. [2]. This method adjusts the input parameter values of the chosen model to minimize the difference between the model output and the measured data [1]. A detailed explanation of the chosen method is provided in Section 2.4.2. The chosen model, the discrete Timoshenko beam model, is described in Section 4.2.

The objective function, which is used to quantify the difference between the model output and the measured data, is indicated in Equation 2.1. The weight factors of the objective function are set to 1, consistent with the studies by Moretti et al. [1] and Taciroglu et al. [2]. Both the model output and the measured data are in the form of natural frequencies and mode shapes, referred to as modal properties. However, compared to the model mode shapes, the measured mode shapes are spatially incomplete. The number of sensors used to obtain the modal displacements is much smaller than the degrees of freedom in the chosen model [7]. Only the modal displacements at the measured heights are known, whereas the model provides the modal displacements at all heights.

Since the mismatch of the mode shapes in the objective function is quantified using the Modal Assurance Criterion (MAC), as indicated in Equation 2.2, and since the MAC requires that the mode shapes be of the same dimension, the mode shapes of the model are reduced to match the dimensions of the measured mode shapes. This is done by considering only the modal displacements of the model corresponding to the measured heights.

The optimization algorithms used in this research are the Sequential Least Squares Quadratic Programming (SLSQP) algorithm, the Differential Evolution (DE) algorithm, and the Particle Swarm Optimization (PSO) algorithm. SLSQP is an individual-based optimization algorithm that uses the gradient of the objective function to guide the search in the solution space [10]. DE and PSO are evolutionary and swarm algorithms, respectively. Details of individual-based optimization algorithms are found in Section 2.3.1. Details of the evolutionary and swarm optimization algorithms are found in Section 2.3.2.

4.2. The discrete Timoshenko beam model

The model used to approximate the dynamic behavior of the two high-rise buildings is a discrete Timoshenko beam model, as illustrated in Figure 4.1. The superstructure is modeled as a discrete Timoshenko beam with bending stiffnesses EI_x and EI_y, shear stiffnesses kG_xA and kG_yA, and masses ρA per unit length.

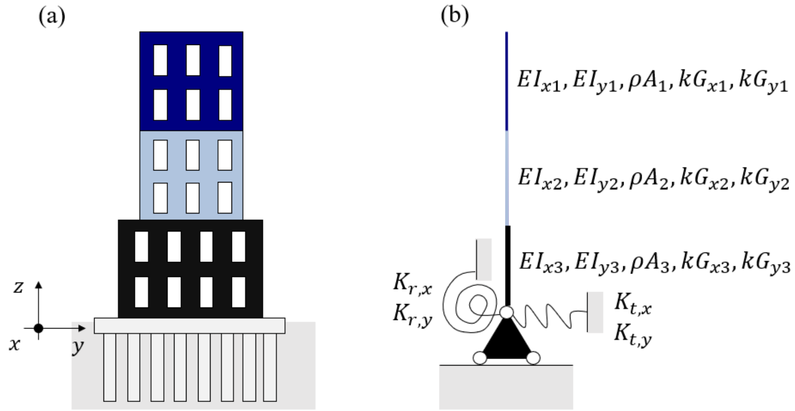

Figure 4.1: A schematic representation of (a) a high-rise building and (b) the discrete Timoshenko beam model used to approximate the dynamic behavior of that building.

The foundation is modeled as a combination of rotational ($k_{r,x}$ and $k_{r,y}$) and translational springs ($k_{t,x}$ and $k_{t,y}$). The model accounts for both bending and shear deformations, and considers irregular stiffness across the height. The model derivation can be found in Appendix A. Its verification can be found in Appendix B. The written Python code can be found in Appendix F.

An important aspect of the model is its frequency spectrum. The frequency spectrum of the discrete Timoshenko beam model is separated in two parts, by a so-called transition frequency, indicated with Equation 4.1:

$$\omega = \sqrt{\frac{kGA}{\rho I}} \tag{4.1}$$

where the eigenmodes below and above this transition frequency exhibit differences in behaviors. Details of the eigenmodes are described in Appendix A.

In the part of the frequency spectrum above this transition frequency, a phenomenon called "the second spectrum of modes" can occur, initially observed by Traill-Nash and Collar [14]. The shapes of these eigenmodes are identical to those appearing in the first part of the spectrum, however, they are associated with higher frequencies.

The findings of Traill-Nash and Collar caused various responses: Abbas and Thomas argued that, aside from a simply supported beam, no second spectrum of modes exists, stating that earlier conclusions about its existence were misinterpretations [9]. Bhashyam and Prathap provided proof of its existence for various boundary conditions, and developed a methodology to categorize the different modes [15]. Levinson and Cooke claimed that, even for the simply supported Timoshenko beam, only one spectrum of modes exists [16].

The ongoing debate may explain why many papers published since 1921 do not provide a complete solution and often fail to address the potential existence of a second spectrum of modes. Despite this, the phenomenon is physically significant because the modes in this spectrum share the same wavelength as the modes of lower frequencies, even if their frequencies differ. This means, for example, that a high-frequency oscillating force could excite and potentially resonate with these modes. Ignoring them could lead to overlooking such phenomena. However, for this research, this is not considered a problem, as the measured modes used are expected to be well below this transition frequency. Details about the phenomenon can be found in Appendix A.

How exactly the discrete Timoshenko beam model is applied to approximate the dynamic behaviors of the residential tower the New Orleans and on the office tower the Delftse Poort, is explained per application in the Chapters 5 and 6.

5
The residential tower New Orleans

To evaluate the effectiveness of vibration-based model updating in estimating the structural properties of a bending-dominant building using a discrete Timoshenko beam model, this chapter applies the technique to the residential tower New Orleans in Rotterdam.

Section 5.1 explains the various studies conducted on the New Orleans. Section 5.2 presents the results of these studies, along with a comparison to the findings from the study by Moretti et al. [1]. Finally, the main findings are summarized in Section 5.3.

5.1. Studies

The research consists of six studies. The first five studies examine how different settings and uncertainties in the model updating process, using a discrete Timoshenko beam model, influence the results obtained. The final study applies model updating to estimate the structural properties of the New Orleans, using the discrete Timoshenko beam model and the measured modal properties. The details of the studies are explained in Sections 5.1.1 to 5.1.6. Appendix D describes how the measured modal properties were obtained.

5.1.1. Influence Features Model

The first study is about the features of the Timoshenko beam model. The model incorporates shear deformations and accounts for irregular stiffness across its height. To assess the influence of these features in the beam model, the natural frequencies and mode shapes obtained with this model are compared with those obtained using the uniform Timoshenko beam and discrete Euler-Bernoulli beam model.

The discrete Timoshenko beam model used in the studies to approximate the dynamic behavior of the New Orleans is created using the Finite Element (FE) model of the building. This FE model, shown in Figure 5.1, is divided into five segments (0 till IV). The segments were defined based on the floor plans. The floors within a segment have the same floor plan in the FE model. Therefore, a five-segment Timoshenko beam model is used to represent the FE model in Figure 5.1.

The initial structural property values applied in this beam model are shown in Table 5.1. These values were obtained using the information of the FE model and the floor plans. Details of the approach with which the structural property values were obtained are described in Appendix C. The second moment of area around the x-axis is denoted as I_x. The rotational stiffness of the foundation around the x-axis is denoted as $K_{r,x}$. The translation stiffness of the foundation in x-direction is denoted as $K_{t,x}$.

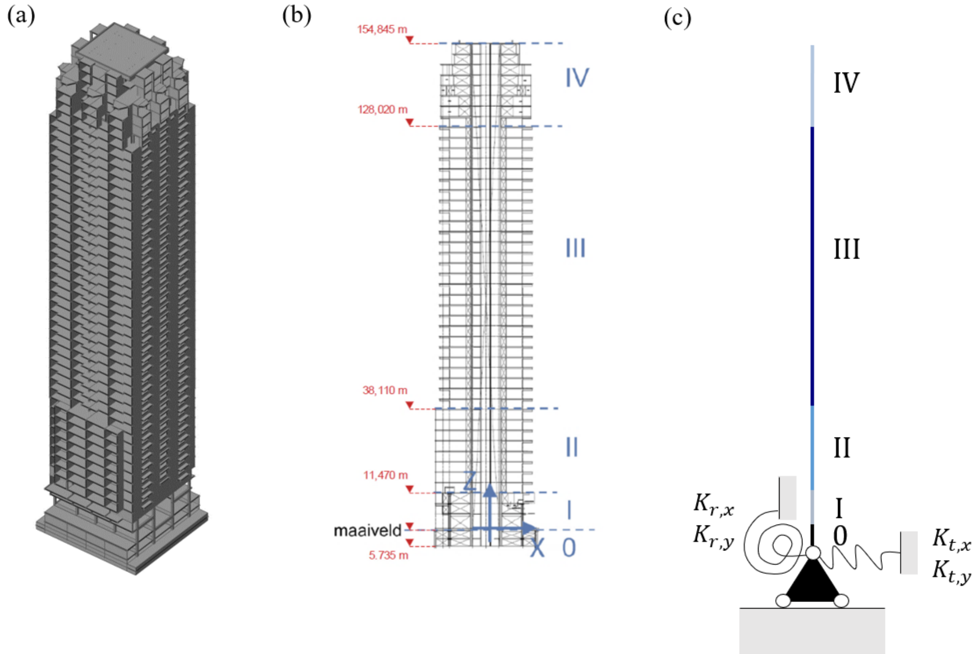

Figure 5.1: Pictures of (a) the FE model of the New Orleans, (b) the 5 segments defined for the New Orleans model, and (c) the five-segment Timoshenko beam model.

Segment	**0**	**I**	**II**	**III**	**IV**
L [m]	5.735	11.47	26.640	89.91	26.825
A[m^2]	784	784	784	784	784
I_x [m^4]	1444	1500	1393	1389	1293
I_y [m^4]	1532	1548	2796	2324	760
ρ [kg/m^3]	1933	495	486	440	383
E [N/m^2]	38.2e9	38.2e9	38.2e9	38.2e9	32.8e9
v [-]	0.2	0.2	0.2	0.2	0.2
G [N/m^2]	15.9e9	15.9e9	15.9e9	15.9e9	13.7e9
k [-]	0.85	0.85	0.85	0.85	0.85
Boundary					
$K_{r,x}$ [GNm/rad]	2625	$K_{t,x}$ [GN/m]	4		
$K_{r,y}$ [GNm/rad]	2380	$K_{t,y}$ [GN/m]	19		

Table 5.1: The structural property values obtained with the floor plans and FE model of the New Orleans.

Obtaining the bending stiffnesses of the segments was challenging. Two approaches were applied to determine the bending stiffness EI_s (with s being x or y) per segment.

The first approach calculated the bending stiffness per segment EI_s using the elastic modulus E applied in the FE model and the second moment of area I, calculated using the term $\frac{1}{12}bh^3$ and the Steiner rule:

$$EI_s = E_s\left(\frac{1}{12}bh^3 + bd^2\right) \tag{5.1}$$

The second approach considered the segments of the FE model. The FE model of the segment was clamped at the bottom, and a force F was applied at the top, resulting in a relative displacement Δ. The bending stiffness EI_s was then determined using this displacement Δ and Equation C.3:

$$EI_s = \frac{FL^3}{3\Delta} \tag{5.2}$$

More information about the applied approaches can be found in Appendix C. The values obtained from each approach are shown in Table 5.2.

Additionally, the displacements at the top, using either a five-segment beam model with these bending stiffnesses or the FE model, are presented, when a force of 10,000 kN is applied and the base of the model is fixed.

	EI$_x$ [Nm^2]		**EI$_y$ [Nm^2]**		**FEM**	
	Steiner	**FE segment**	**Steiner**	**FE segment**	**Y**	**X**
0	3.19e13	2.37e13	6.71e13	5.87e13		
I	3.31e13	3.62e13	6.77e13	3.66e13		
II	3.08e13	1.04e14	1.22e14	4.76e13		
III	3.07e13	8.92e13	1.04e14	4.59e13		
IV	2.45e13	2.97e13	2.85e13	2.73e13		
Δu [mm]	887	472	344	613	516	394

Table 5.2: The bending stiffnesses and the total displacements determined with the different approaches.

While both approaches offer ways to determine the bending stiffness EI_s, both have limitations in accurately estimating it. With the Steiner approach, the obtained displacement in the y-direction (Δ u = 887 mm) showed significant differences with the displacement of the FE model (Δ u = 516 mm). This is suspected to be due to the exclusion of certain columns and walls in the calculation of the second moment of area I_x. These are labeled as C1 and C2 in Figure C.2 and Figure C.3, and as W5 in Figure C.4, Figure C.5, and Figure C.6. For illustration, Figure C.3 is shown as Figure 5.2. Figure C.5 is shown as Figure 5.3.

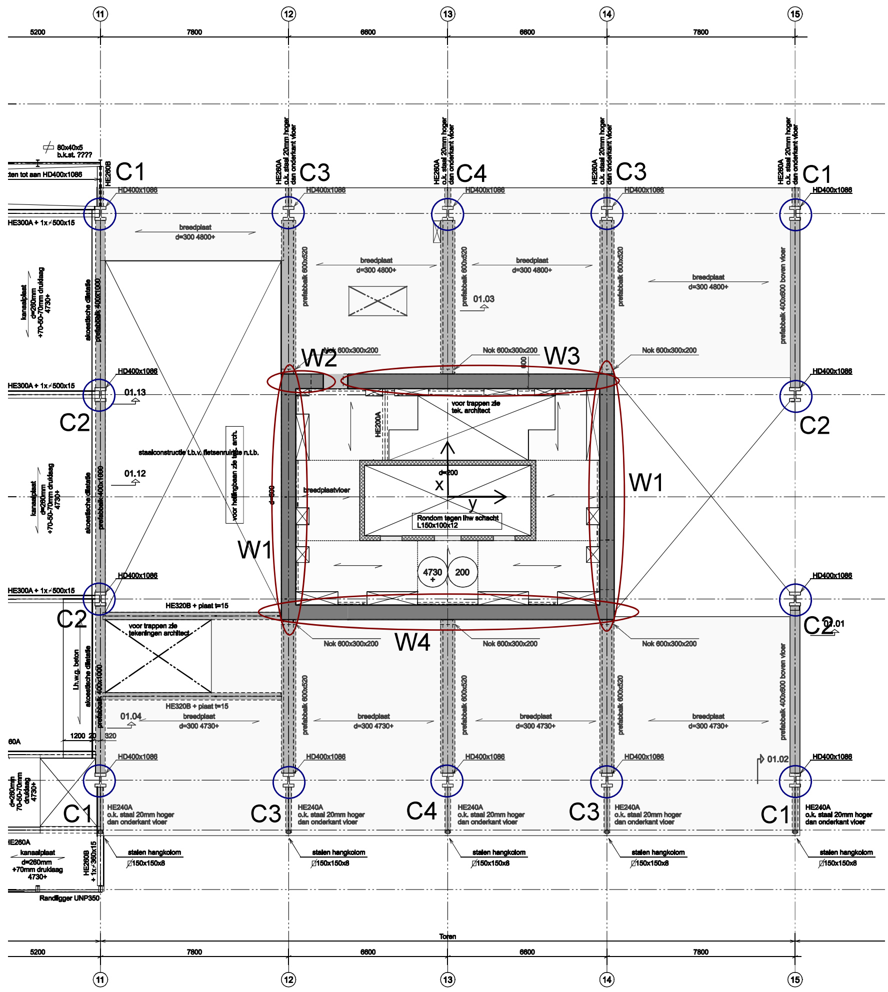

Figure 5.2: The floor plan of segment I.

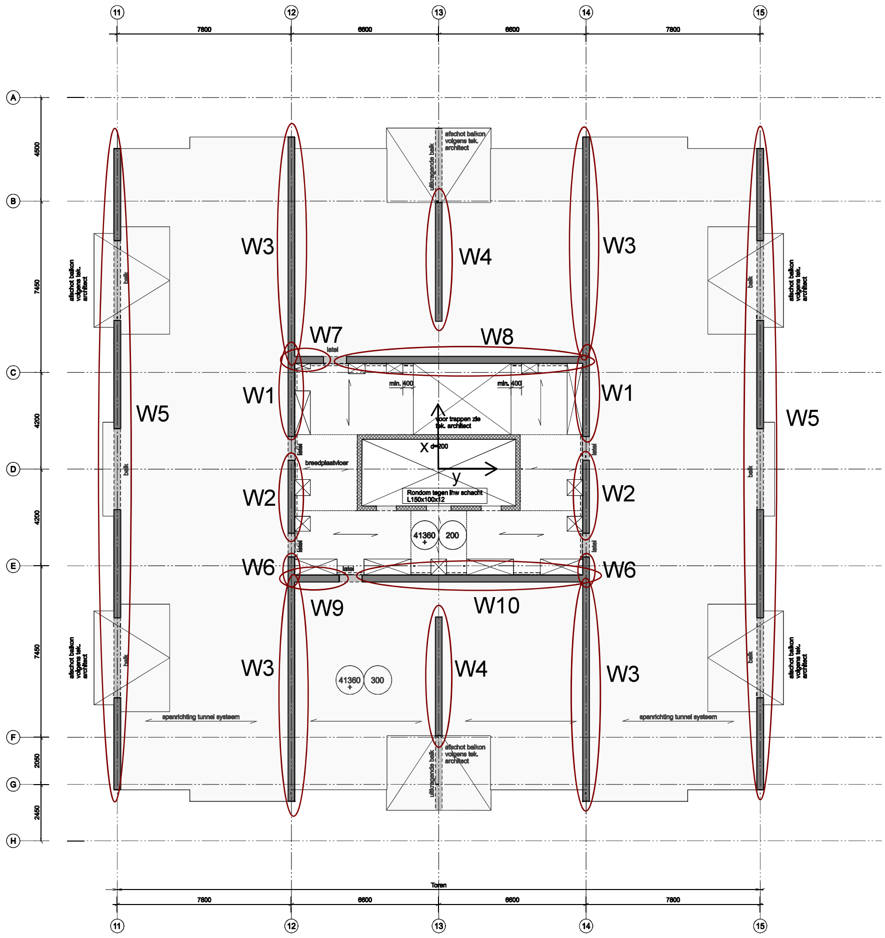

Figure 5.3: The floor plan of segment III.

In the Steiner approach, full rigid connections are assumed between the structural elements. Due to the lack of a rigid connection between the core walls and these elements, it was decided not to include them in the calculation. However, despite the absence of a rigid connection, it is reasonable to expect that these elements still contribute partially to the rotational resistance, as they are connected to the core through the floor. The actual contribution of these elements lies somewhere between the extremes of full inclusion and exclusion, but this contribution cannot be calculated using Steiner.

With the FE segment approach, as shown in Table 5.2, the bending stiffnesses EI_x vary significantly. This is unexpected, as the floor plans between the segments do not change substantially. This approach fails to account for the intermediate relationships between the different segments (0 through IV) of the building. For example, segments 0 and I have column structures (as shown in Figure 5.2 for segment I), which exhibit weaker behavior when considered individually than when integrated into the overall system. This is due to the additional mass from other segments. Therefore, considering these segment individually will lead to an inaccurate assessment of their bending stiffnesses.
Therefore, the bending stiffnesses of the Steiner approach are applied, with a correction factor included to account for the displacement differences between the FE model and five-segment beam using these segment stiffnesses. Since the FE model is stiffer in the y-direction, the segment stiffnesses EI_x are multiplied by $\frac{887}{516}$. Since the FE model is weaker in the x-direction, the segment stiffnesses EI_y are multiplied by $\frac{344}{394}$.

It is assumed that the ratios of the initial structural properties between the segments are well estimated with the Steiner approach. By maintaining these ratios, only one set of parameters of the superstructure needs to be updated, thereby reducing the total number of parameters that needs to be updated.

The uniform Timoshenko beam model and the discrete Euler-Bernoulli beam model used in this study are shown in Figure 5.4. For the discrete Euler-Bernoulli beam, the same initial structural properties are applied as those for the five-segment Timoshenko beam model. For the uniform Timoshenko beam model, the volume weighted averages of these values are used.

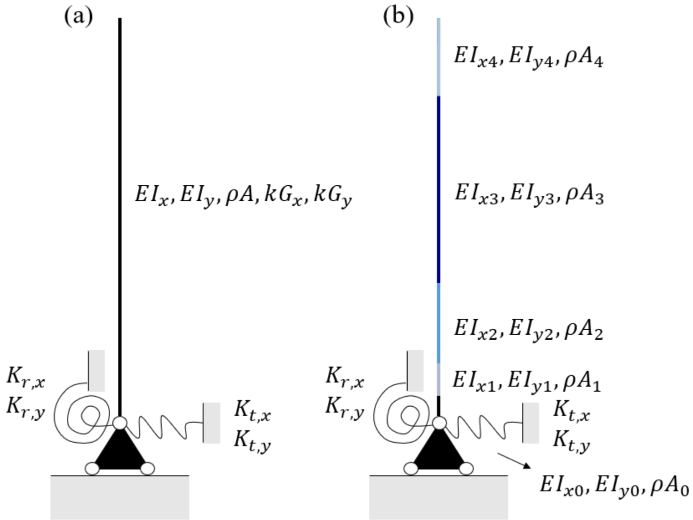

Figure 5.4: Pictures of (a) the uniform Timoshenko beam model (b) the discrete Euler-Bernoulli beam model.

5.1.2. Influence Parameters

The second study investigate the influence of the input parameters of the model. With model updating, a decision must be made about which parameters to update. To understand how each input parameter of the discrete Timoshenko beam influences the modal properties of the model (i.e., natural frequencies and mode shapes), a sensitivity study is conducted where the values of each input parameter are varied within a specified range.

The range over which each input parameter is varied corresponds to its initial structural property value in Table 5.1, scaled by a factor of ten, as shown for the parameter kG in Figure 5.5. This range is chosen to ensure that a sufficiently broad range is investigated. Parameters A and ρ, representing the cross-section and density, respectively, are excluded from this study, as these parameters do not address the model uncertainties.

5.1.3. Influence Optimization Algorithm

In the third study, the influence of the optimization algorithm on model updating with a discrete Timoshenko beam model is investigated through a convergence study. The study uses three different algorithms on the same case: Sequential Least Squares Quadratic Programming (SLSQP), Differential Evolution (DE), and Particle Swarm Optimization (PSO). For each algorithm, different amounts of candidate solutions are considered. The updating parameters and the value ranges considered for these updating parameters are based on the study ”Influence Parameters” in Section 5.1.2.

To conduct this study, measured modal properties of a building with known structural properties are required. Since such data is not available, as the structural properties of the New Orleans are not known, a numerical case is created using the structural property values from Table 5.1. Random scalars are applied to the initial values in Tables 5.1 in both x- and y-direction, creating another model with known structural and modal properties.

The random scalars applied are listed in Table 5.3. The modal properties of this new model serve as the measured modal properties for the study. The number of modal properties used to fit this model aligns with the amount of measured modal properties of the New Orleans building: two bending modes in the x-direction and three bending modes in the y-direction.

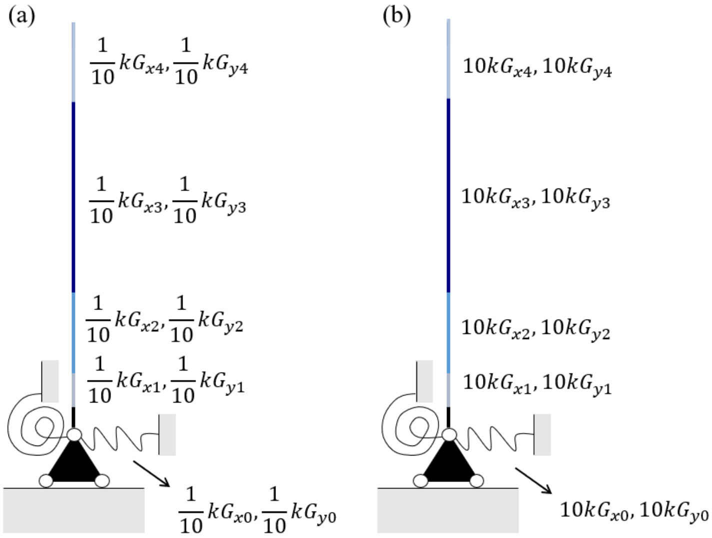

Figure 5.5: Pictures (a,b) of parameter kG scaled by a factor of 10.

The modal displacements considered are corresponding to the measured heights of the New Orleans (51.4 m, 114.6 m and 147.9 m).

It should be noted that the random scaling factors are applied exclusively to the parameters selected for updating. Otherwise, it would be impossible to completely match the model output with the measured modal properties after model updating.

	K_r	K_t	E	I	ρ	kG
Scalar (-)	0.4	9.0	7.0	1.5	2.0	0.2

Table 5.3: The random scalar applied on the initial structural properties values of Table 5.1.

5.1.4. Influence Amount of Measured Modal Properties

In the fourth study, the influence of the amount of measured modal properties on model updating with the discrete Timoshenko beam model is examined. A convergence study is conducted that considers an increasing number of modes on the same case:

- The first bending mode in the x- and y-direction
- The first two bending modes in the x- and y-direction
- The first two bending modes in the x-direction and the first three bending modes in the y-direction.

Different numbers of candidate solutions are considered. The updating parameters and the value ranges considered for these updating parameters are based on the study ”Influence Parameters” in Section 5.1.2. The algorithm applied in the model updating process is determined using the study ”Influence Optimization Algorithm” in Section 5.1.3. Since this study also requires measured modal properties of a building with known structural properties, the numerical case from the study ”Influence Optimization Algorithm” is also applied here.

5.1.5. Influence Measurements Uncertainties

The fifth study investigates the impact of errors in the measured modal properties (i.e., the measured natural frequencies and mode shapes) on model updating with the discrete Timoshenko beam model. The sensitivity study introduces random errors into the measured modal properties to understand how these uncertainties affect the results obtained. The errors applied to the natural frequencies range up to 10%, and the errors applied to the mode shapes range up to 20%. An example of an error distributions is illustrated in Figure 5.6 for the first natural frequency and mode shape.

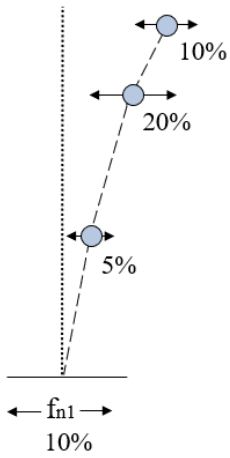

Figure 5.6: A visualization of the random errors applied on the first natural frequency and mode shape.

The updating parameters and the value ranges considered for these updating parameters are based on the study "Influence Parameters" in Section 5.1.2. The algorithm applied in the model updating process is determined using the study "Influence Optimization Algorithm" in Section 5.1.3. The amount of candidate solutions considered is chosen based on the study "Influence Amount of Measured Modal Properties" in Section 5.1.4.

Since this study also requires measured modal properties of a building with known structural properties, the numerical case from the study "Influence Optimization Algorithm" is applied here as well.

5.1.6. Estimating Structural Properties

This study applies model updating on the New Orleans to estimate its structural properties using the discrete Timoshenko beam model and the measured modal properties. The measured modal properties of the New Orleans are indicated in Table 5.4.

Mode	**1**	**2**	**3**	**4**	**5**
Dominant direction	X	Y	Y	X	Y
Natural frequency [Hz]	0.282	0.291	1.332	1.527	2.771
Modal Displacement					
Height 1 (51.4 m)	−0.30	−0.29	−0.90	0.89	−0.73
Height 2 (114.6 m)	−0.71	−0.78	0.19	−0.23	0.83
Height 3 (147.9 m)	−1.00	−1.00	1.00	−1.00	−1.00

Table 5.4: The measured natural frequencies and modal displacements of the residential tower New Orleans.

To determine these modal properties, acceleration sensors were installed on the 15th, 34th, and 44th floors. The positions and orientations of the sensors are illustrated in Figure 5.7. Details of the data analysis applied to obtain the measured modal properties are described in Appendix D. The updating parameters and their value ranges considered are based on the study "Influence Parameters" in Section 5.1.2. The algorithm applied in the model updating process is determined using the study "Influence Optimization Algorithm" in Section 5.1.3. The amount of measured properties from Table 5.4 used to fit the model and the amount of candidate solution considered are based on the study "Influence of the Amount of Measured Properties" in Section 5.1.4.

The model updating process follows the approach described by Moretti et al. [1] and consists of two parts. In the first part, model updating is carried out using the settings specified above. However, unlike the method used by Moretti et al. [1], which retains the lowest quantile of the candidate solutions (i.e., the quantile that best fit the measured modal properties after updating), this study retains the

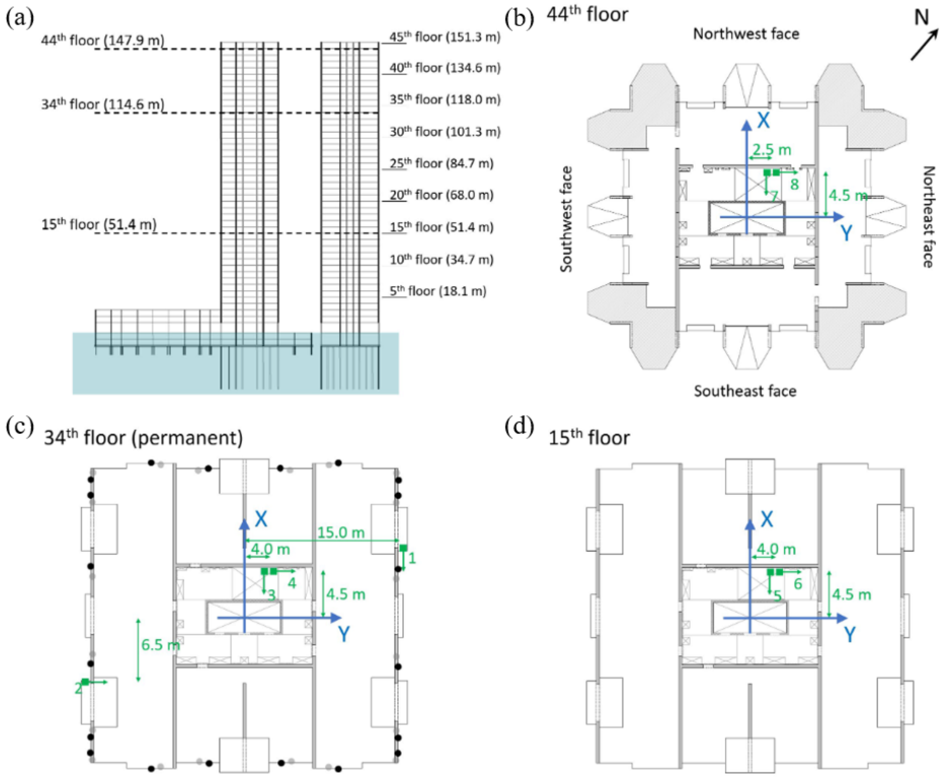

Figure 5.7: An schematic (a) of the New Orleans and the positions of the acceleration sensors (green squares) on (b) the 44th floor, (c) the 34th floor, and (d) the 15th floor. The green arrows indicate the directions of the acceleration sensors [17, 12].

lowest octile of the candidate solutions. This modification is done because the discrete Timoshenko beam model is expected to be more complex than the uniform Euler-Bernoulli beam model, resulting in a more complex solution space for identifying the global minimum. New structural property value ranges for the updating parameters are defined based on the minimum and maximum values of these retained solutions. Using these newly refined ranges, another selection and model updating process is carried out (the second part) in the same manner as the first part. The reasons behind the choices made in this approach by Moretti et al. [1] are unclear. This approach was chosen for this research to ensure a fair comparison between the findings of the study conducted by Moretti et al. [1] and this research.

5.2. Results

In this section, the results of the research conducted are presented and discussed. Sections 5.2.1 to 5.2.5 present the results of the studies where the influence of various settings and uncertainties in model updating with a discrete Timoshenko beam model are investigated. Section 5.2.6 show the results of applying model updating to estimate the structural properties of the New Orleans.

5.2.1. Influence Features Model

To begin, the results of the first study are presented. The obtained natural frequencies using different beam models are shown in Tables 5.5 and 5.6. Their mode shapes are illustrated in Figures 5.8 and 5.9. The modal properties of the Finite Element (FE) model and the measured modal properties are also shown as reference.

Figures 5.8 and 5.9 show that the mode shapes of the FE model for the two lowest bending modes in both directions closely resemble the measured ones, indicating a good representation of reality. However, the mode shape of the third bending mode does not align closely with the measured third bending mode. The FE model remains a simplified beam model. Factors such as simplifications in boundary conditions, insufficient mesh refinement, or modeling assumptions can cause the differences observed.

Tables 5.5 and 5.6 show that the natural frequencies of the FE model tend to differ more from the measured natural frequencies than those of the discrete Timoshenko beam model. This is unexpected, as the FE model incorporates more realistic boundary conditions and better captures complex geometries compared to the discrete Timoshenko model. However, it is unlikely that the discrete Timoshenko beam model provides a more accurate approximation of the high-rise building than the FE model. The natural frequency of the third bending mode and the mode shapes of the higher bending modes in the y-direction of this beam model exhibit significant discrepancies from the measured data.

The discrete Timoshenko beam model and the Euler-Bernoulli beam model show similarities in their natural frequencies, indicating that shear effects have minimal impact on the overall behavior of the structure. However, Figure 5.8 shows that shear effects are significant in describing the mode shape of the second bending mode in the x-direction. The uniform Timoshenko beam model shows comparable results with the discrete Timoshenko beam model. This suggest that the initial values of the parameters of the different segments are similar. For the second and third bending mode in the y-direction, the FE model exhibits greater translation at the lower boundary compared to the beam models. Further investigation is required to identify the cause of this discrepancy.

Measured [Hz]		**FEM (Δ_{measured} %) [Hz]**		**Discrete Timoshenko beam model (Δ_{measured} %) [Hz]**	
X	**Y**	**X**	**Y**	**X**	**Y**
0.28	0.29	0.23 (−17.9)	0.20 (−31.0)	0.25 (−10.7)	0.23 (−20.7)
1.53	1.33	1.25 (−18.3)	1.02 (−23.3)	1.48 (−3.3)	1.43 (+7.5)
	2.77		2.14 (−22.7)		3.78 (+36.5)

Table 5.5: The measured natural frequencies, the natural frequencies of the FEM, and the natural frequencies of the discrete Timoshenko beam model.

Discrete Euler-Bernoulli beam model (Δ_{measured} %) [Hz]		**Uniform Timoshenko beam model (Δ_{measured} %) [Hz]**	
X	**Y**	**X**	**Y**
0.22 (−21.4)	0.21 (−27.6)	0.21 (−25.0)	0.19 (−34.5)
1.49 (−2.6)	1.38 (+3.8)	1.30 (−15.0)	1.26 (−5.3)
	3.75 (+35.4)		3.48 (+25.6)

Table 5.6: The natural frequencies of the discrete Euler-Bernoulli beam model and the uniform Timoshenko beam model.

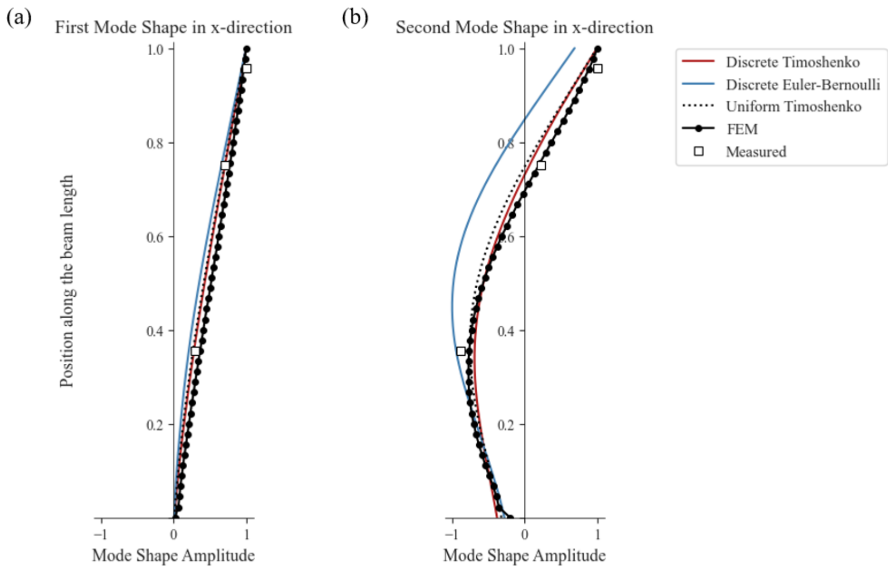

Figure 5.8: The measured and model mode shapes in x-direction

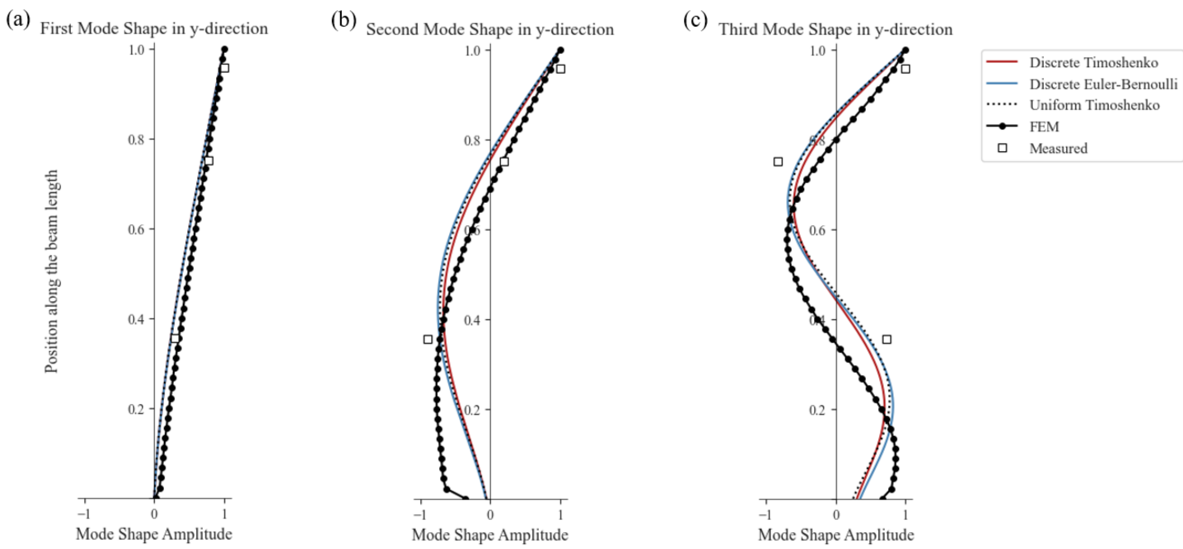

Figure 5.9: The measured and model mode shapes in y-direction

5.2.2. Influence Parameters

This section presents the results of second study, which changed the input parameter values of the discrete Timoshenko beam model to see their influence on the modal properties. The results of parameters K_r and K_t in the y-direction are shown in Figures 5.10 and 5.11. The results of parameters E and I in the y-direction are shown in Figures 5.13 and 5.12. The result of parameter kG in the y-direction is shown in Figure 5.14. The change in the mode shape is quantified using the MAC (Equation 2.2).

According to Figure 5.10, adjusting the value of parameter K_r primarily influence the first natural frequency. Changes in K_r do not affect the mode shapes. In contrast, Figure 5.11 shows that parameter K_t only affects the higher bending modes.

Parameters E and I have the same influence on the bending modes. These parameters often occupy similar positions within the model, leading to comparable influences on the natural frequencies and mode shapes. Due to these similarities between parameters E and I, it is chosen to consider the updating results of these parameters together as EI for the rest of the research. Figure 5.14 shows that parameter kG does not affect the model output. This is in line with the results obtained in Table 5.5 in Section 5.2.1, which indicated that shear effects have minimal impact on the overall behavior of the New Orleans. Therefore, kG is not selected as an updating parameter.

The results in the x-direction are comparable to the results found in the y-direction. Therefore, only the results in y-direction are shown. The results in x-direction are shown in Appendix E. Based on the results, it is chosen to update parameters K_r, K_t, E, and I in the following studies of the research.

To summarize, it should be noted that parameter K_r primarily influence the first natural frequency, and parameter K_t affects the higher bending modes. Parameters E and I exhibit similar influences, as the parameters often occupy similar positions within the model. Therefore, the results for these parameters will be considered together as EI for the rest of the research. Parameter kG is not selected, as the modal properties of the model are insensitive to this parameter.

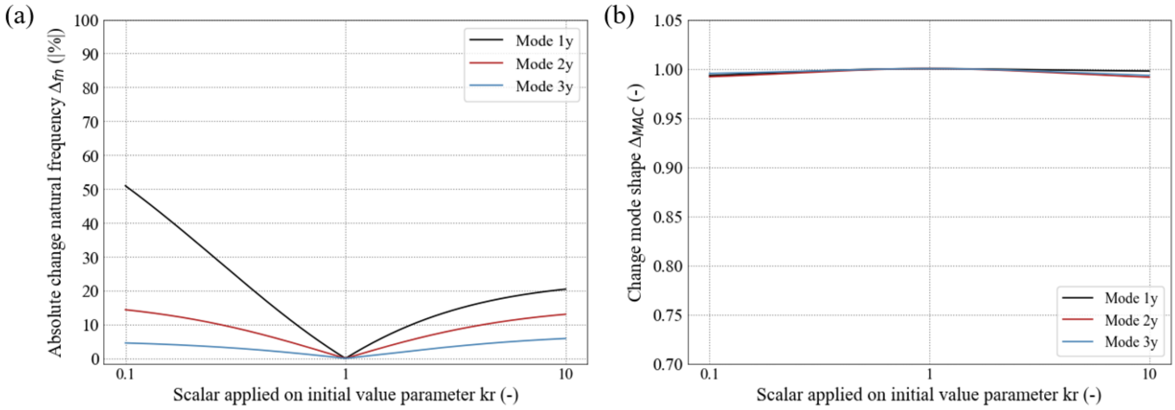

Figure 5.10: The influence of parameter K_r on the model output in y-direction.

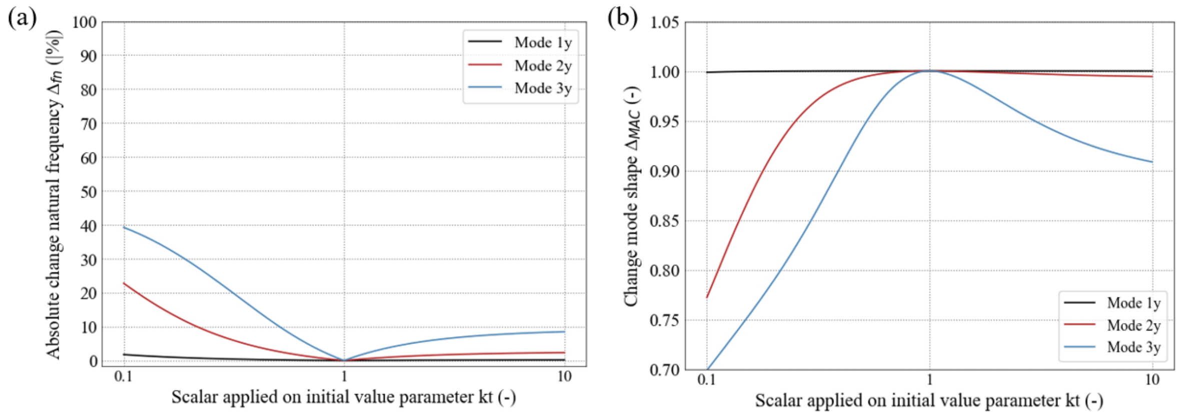

Figure 5.11: The influence of parameter K_t on the model output (a,b) in y-direction.

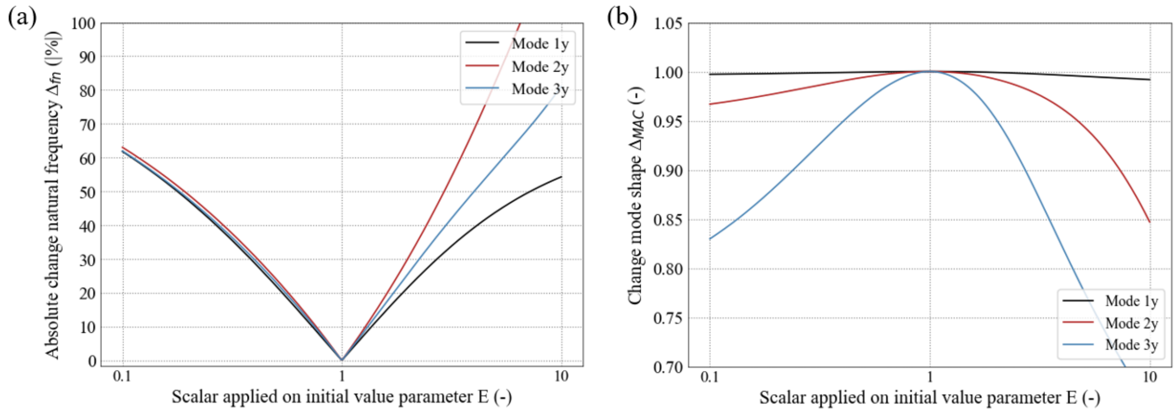

Figure 5.12: The influence of parameter E on the model output (a,b) in y-direction.

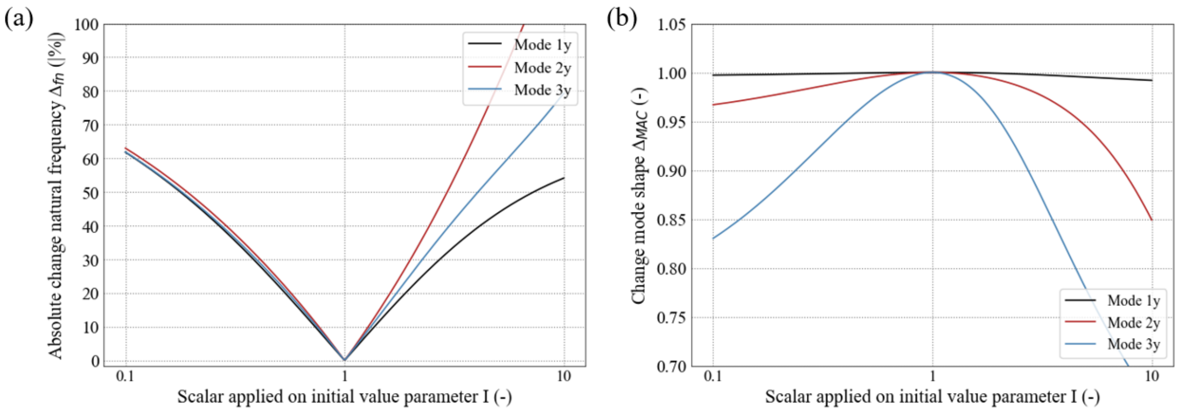

Figure 5.13: The influence of parameter I on the model output (a,b) in y-direction.

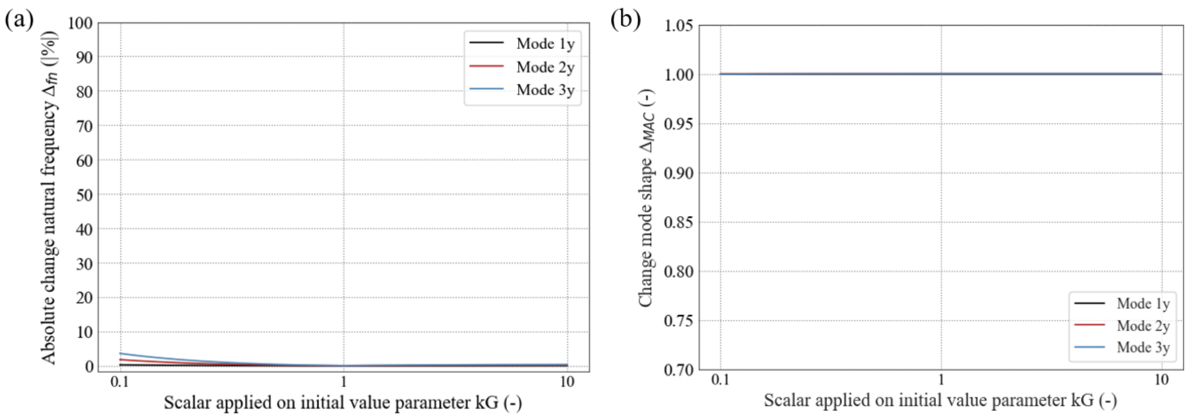

Figure 5.14: The influence of parameter kG on the model output (a,b) in y-direction.

5.2.3. Optimization Algorithm

In this section, the results of the third study are shown, which applied different optimization algorithms on the same numerical case. The updating parameters were K_r, K_t, E and I, which were updated at the same time.

To create this numerical case, random scalars were applied to the initial values of the chosen updating parameters, indicated in Table 5.1, to generate another model with known structural properties and mode shapes. The scalars applied are indicated in Table 5.7. The modal properties of this new model, indicated in Table 5.8, were selected as measured modal properties for this study.

	K_r	K_t	E	I
Scalar (-)	0.4	9.0	7.0	1.5

Table 5.7: The random scalar applied on the initial structural properties values of Table 5.1.

Mode	**1**	**2**	**3**	**4**	**5**
Dominant direction	X	Y	Y	X	Y
Natural frequency [Hz]	0.226	0.235	3.83	4.345	11.132
Modal Displacement					
Height 1 (51.4 m)	0.36	0.36	−0.75	−0.78	0.39
Height 2 (114.6 m)	0.78	0.78	0.10	0.13	−0.57
Height 3 (147.9 m)	1.00	1.00	1.00	1.00	1.00

Table 5.8: The modal properties of the numerical case.

Different amounts of candidate solutions were considered, and for each amount, the candidate solution that best fitted the measured modal properties after model updating is shown. To maintain the compactness of the results, the results are presented as ratios relative to their initial values in Table 5.1. For example, a value of 0.8 indicates that after the model updating, the new value of the structural property is 80% of its initial value. Therefore, if the model updating was successful with the chosen algorithm, the results should match the scalars listed in Table 5.7. The results of parameters K_r and K_t are shown in Figures 5.15 and 5.16. The results of parameters E and I are shown together as EI in Figure 5.17.

According to the Figures, K_r, K_t and EI are well estimated by all algorithms. The differences between the algorithms are small. The SLSQP algorithm achieved the most accurate estimation by all parameters. Therefore, for the following studies, the SLSQP algorithm is selected for model updating.

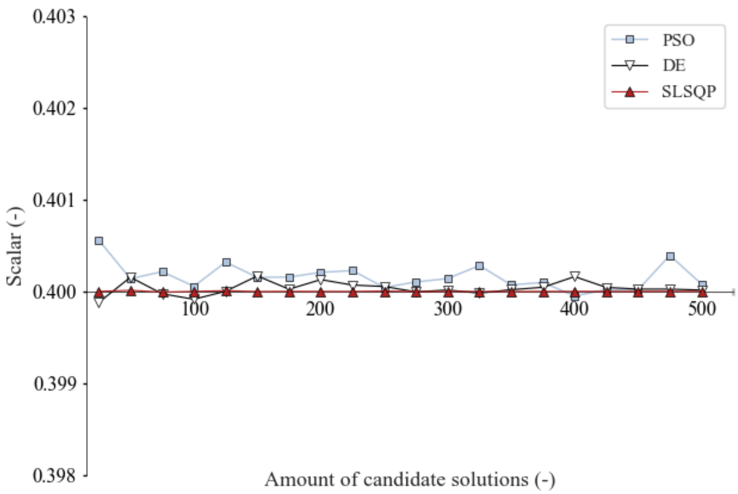

Figure 5.15: The influence of the different algorithms on the model updating result of parameter K_r.

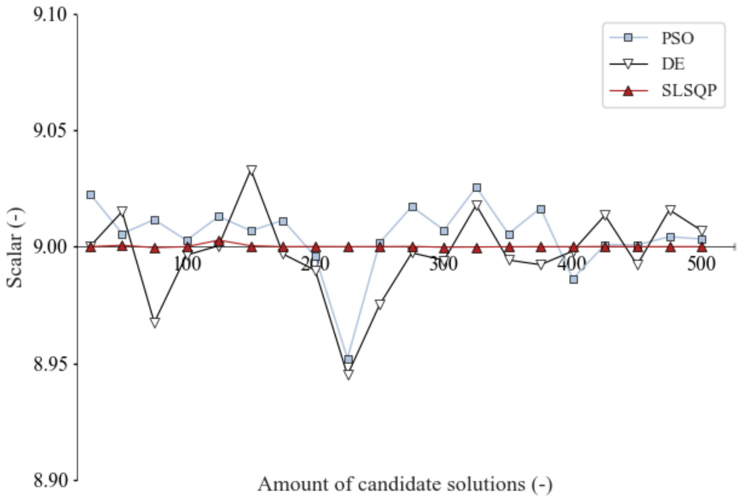

Figure 5.16: The influence of the different algorithms on the model updating result of parameter K_t.

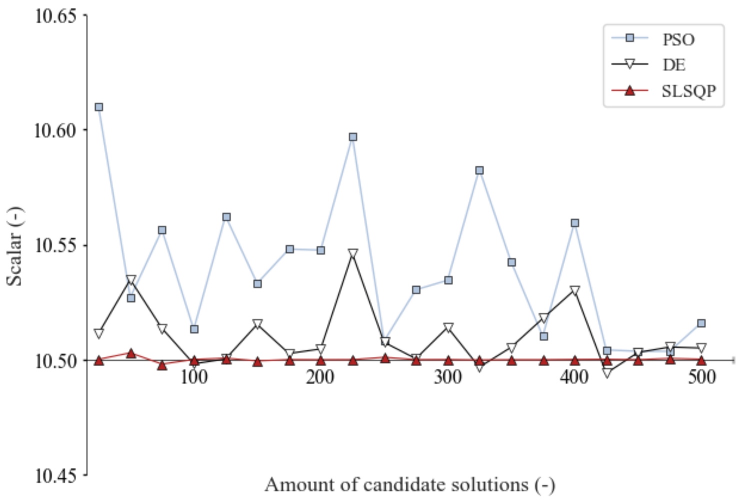

Figure 5.17: The influence of the different algorithms on the model updating result of the product EI.

5.2.4. Influence Amount of Measured Modal Properties

This section presents the model updating results of the fourth study, which applied different amount of measured modal properties on the same numerical case. The parameters chosen to update in this study were K_r, K_t, E and I, which were updated at the same time. The SLSQP was chosen as optimization algorithm.

This study used the same numerical case described in the study "Influence Optimization Algorithm" in Section 5.1.3. Per amount of measured modal properties, different amounts of candidate solutions were considered, and for each amount of candidate solutions, only the candidate solution that best fitted the measured modal properties after updating is shown.

The results are presented as ratios relative to their initial values in Table 5.1. Therefore, if model updating was successful with the amount of measured modal properties, the results should match the scalars listed in Table 5.7. The results of the parameters K_r and K_t are shown in Figures 5.18 and 5.19. The result of EI is shown in Figure 5.20.

Figure 5.18 shows that it is sufficient to estimate parameter K_r by considering only the lowest bending mode in both the x- and y-directions. This is in line with the influence observed in Figure 5.10 in Section

5.2.2, which indicated that this parameter mainly affects the first natural frequency. However, Figures 5.19 and 5.20 indicate that considering only the lowest bending mode in both the x- and y-directions is insufficient to estimate K_t and EI. For parameter K_t, this is in line with Figure 5.11 in Section 5.2.2, which shows that parameter K_t has a insensitivity to the first bending mode. However, Figures 5.12 and 5.13 shows that parameters E and I do not exhibit insensitivity to the first bending mode. It is suspected that the first bending mode alone does not provide enough information to estimate parameters E and I. Higher bending modes, which are more sensitive to the bending stiffness, are needed, as they provide more comprehensive information.

The difference in the results K_r, K_t, EI when considering two or three bending modes in y-direction is small. Therefore, both scenarios are considered in the study ”Estimating Structural Properties”, using 400 candidate solutions. For the study ”Influence Measurement Uncertainties”, 200 candidate solutions are considered.

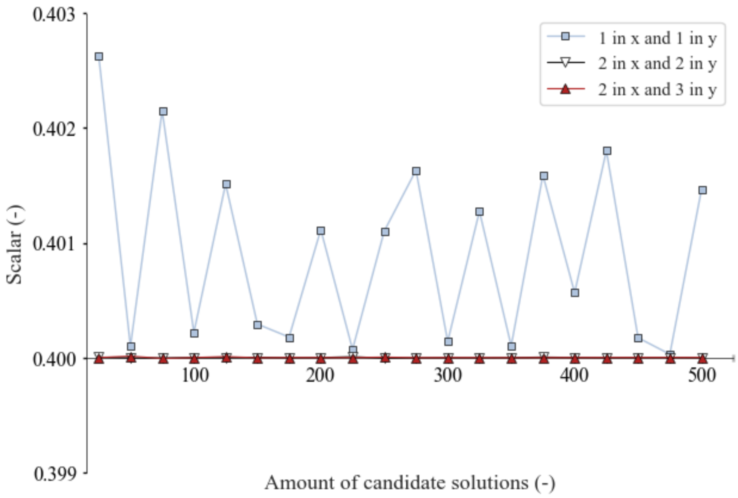

Figure 5.18: The influence of the amount of modal information on the model updating result of parameter K_r.

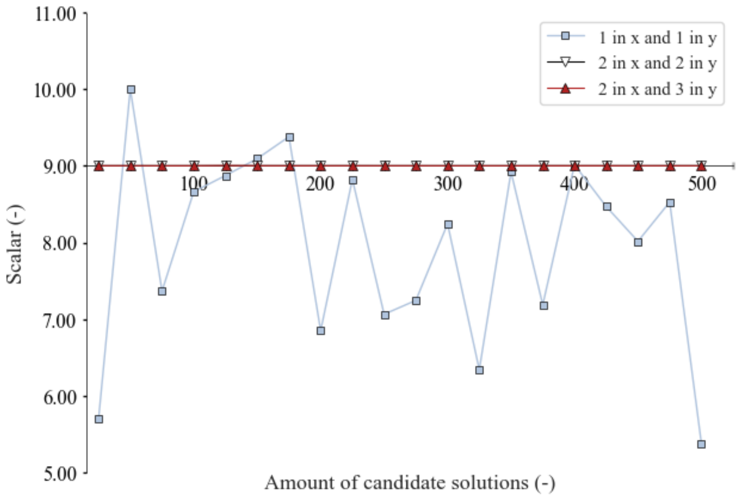

Figure 5.19: The influence of the amount of modal information on the model updating result of parameter K_t.

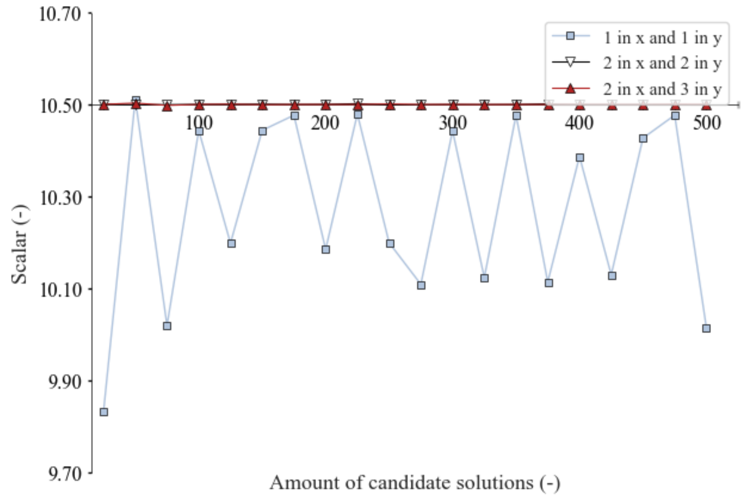

Figure 5.20: The influence of the amount of modal information on the model updating result of EI.

5.2.5. Measurement Uncertainties

This section shows the results of the fifth study, which applied errors to the measured modal properties to see their influence on the results obtained. The parameters updated in this study were K_r, K_t, E and I. These were updated at the same time, using the SLSQP optimization algorithm. The study used the same numerical case described in the study ”Influence Optimization Algorithm” in Section 5.1.3.

Two hundred candidate solution were considered, and only the candidate solution with the lowest objective function value (i.e., the solution that best fitted the measured modal properties after updating) is shown. As adding random errors to the measured modal properties and conducting model updating was repeated ten times, ten solution are shown in total. The results are presented as ratios relative to the initial values shown in Table 5.1. Therefore, if the model updating was insensitive to the errors applied, the results should match the scalars listed in Table 5.7.

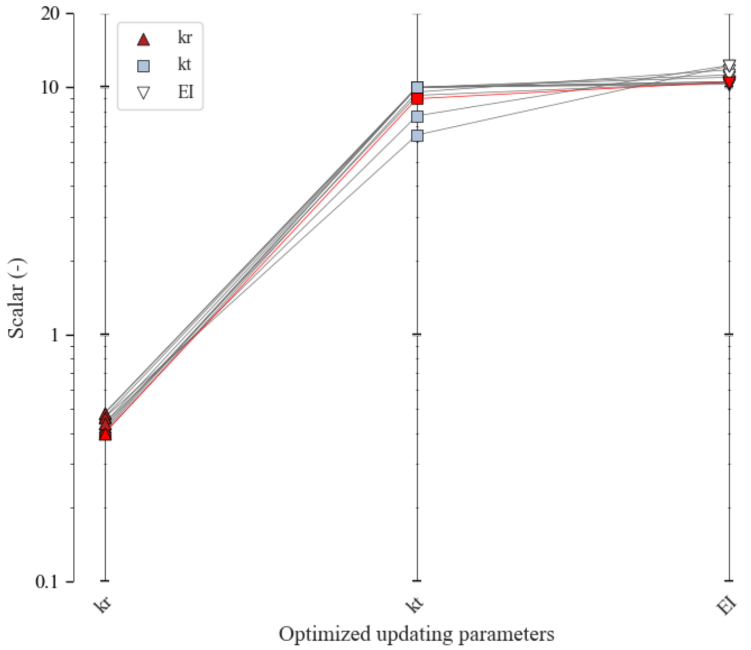

Figure 5.21: The influence of measurement uncertainties on the model updating results.

The results of K_r, K_t, EI are shown in Figure 5.21. The values of the different parameters that belong together are connected with grey lines. The red solution indicates the scalars of Table 5.7.

Parameter K_r shows limited sensitivity to the uncertainties in the measurements, with values obtained between 0.40 and 0.47. Parameters K_t and EI, however, show significant sensitivity to the uncertainties. The values of parameter K_t range between 6.41 and 10.0. EI shows values ranging between 10.29 and 12.28.

5.2.6. Estimating Structural Properties

To conclude, the results using the measured properties of the New Orleans are shown. The parameters updated in this study were K_r, K_t, E and I, using the SLSQP algorithm. Four hundred candidate solutions were considered, and the lowest octile of these solutions, i.e., the octile with the smallest objective function values, were retained.

Table 5.9 presents the estimated and measured natural frequencies and the Modal Assurance Criterion (MAC) values between the estimated and measured mode shapes. Case 1 indicates the results of the model updating using the two lowest bending modes in both x- and y-directions, and Case 2 indicates the results of the model updating using two bending modes in the x-direction and three bending modes in the y-direction.

Mode [-]	Natural frequency [Hz]				MAC [-]	
	Design	**Measured**	**Case 1** (Δ_{measured} %)	**Case 2** (Δ_{measured} %)	**Measured vs. Case 1**	**Measured vs. Case 2**
1 (x)	0.200	0.282	0.282 (0%)	0.282 (0%)	0.998	0.998
2 (x)	1.020	1.527	1.527 (0%)	1.527 (0%)	0.999	0.999
1 (y)	0.230	0.291	0.291 (0%)	0.241 (-17.2%)	0.999	0.995
2 (y)	2.140	1.332	1.332 (0%)	1.332 (0%)	1.000	0.963
3 (y)	2.440	2.771	-	2.773 (+0.1%)	-	0.872

Table 5.9: The measured and estimated natural frequencies and MAC values between the measured and estimated mode shapes.

In Case 1, the median natural frequencies closely match the measured natural frequencies. However, in Case 2, there are differences in the estimated and measured first natural frequency. Adjustments to the values of the structural properties to better match the third bending mode negatively affected the estimation of the first natural frequency. Moreover, the MAC values in Case 2 indicate that the measured third bending mode shape is not accurately described. Despite incorporating additional features compared to the model used in the study by Moretti et al. [1], the discrete Timoshenko beam model is still insufficient for accurately describing the third bending mode. Therefore, the obtained structural properties values of Case 2 are not considered.

Figures 5.22 and 5.23 illustrate the estimated structural property values in the x- and y-directions obtained through model updating for Case 1. The results are presented as ratios relative to the initial values shown in Table 5.1. The values of the different parameters that belong together are connected with grey lines. For the estimates per parameter, the Coefficient of Variation (CV) is calculated. The CV shows the relative variability of the estimates:

$$\text{CV} = \left(\frac{\sigma}{\mu}\right) \times 100\% \tag{5.3}$$

where σ is the standard deviation and μ is the mean. A lower CV indicates less variability relative to the mean, suggesting more consistency in the estimates. A higher CV indicates greater dispersion in the estimates. The distributions of $K_{r,x}$ and $K_{r,y}$ exhibit low Coefficients of Variation (CVs), of 4.54% and 6.91%, respectively, suggesting that the parameters should remain around their initial values. This is a improvement compared to the study by Moretti et al. [1], as in that study, the estimations of the structural property $K_{r,y}$ showed high uncertainty (a Coefficient of Variation of 49.6%) after model updating. Similarly, the distributions of $K_{t,x}$ and $K_{t,y}$ exhibit low CVs of 2.27% and 5.87%, respectively. These distributions suggest that their values should be lower than the design values, although there is a noticeable magnitude difference in these estimates. The distribution of EI_x and EI_y show also low uncertainties, indicated with Coefficients of Variation (CVs) of 5.06% and 15.45% respectively, suggesting that the bending stiffness should be higher. Based on these results, it can be stated that estimating the structural parameters of the New Orleans using vibration-based model updating was effective.

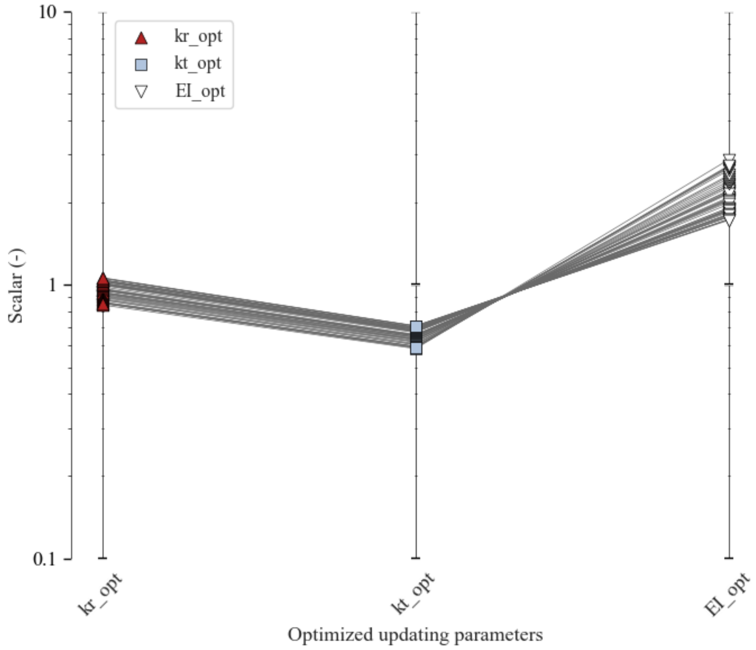

Figure 5.22: The estimated parameter values of Case 1 in x-direction.

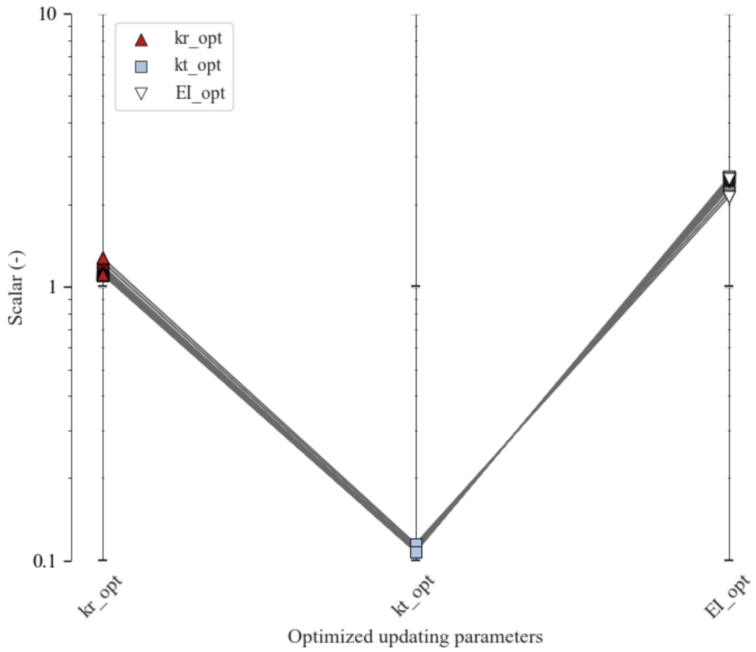

Figure 5.23: The estimated parameter values of Case 1 in y-direction.

5.3. Main Findings

With the research, several studies were conducted to evaluate the influence of certain settings and uncertainties in model updating and to assess the effectiveness of the technique in estimating the structural properties of a bending-dominant building. The main findings were:

- **Setup Models**: Obtaining values for the bending stiffness of the segments using the FE model proved to be difficult. he Steiner approach, which calculated the bending stiffness per segment using the elastic modulus E and the second moment of inertia I (calculated with $\frac{1}{12}bh^3$ and the Steiner rule), assumed fully rigid connections between elements when determining I, making it not possible to accurately account for the contributions of elements that were not rigidly connected.
  The FE approach, which used the relative displacement Δ of the FE segment to calculate the bending stiffness, did not account for the intermediate relationships between segments. Segments 0 and I had column structures, which exhibited weaker behavior when considered individually compared to when they were part of the overall system. This was due to the additional mass of the other segments. Therefore, considering these segments individually led to an inaccurate assessment of their bending stiffnesses.
- **Influence of Parameters**: Parameter K_r showed primarily influence on the first natural frequency, and parameter K_t showed primarily influence on the higher bending modes. Parameters E and I exhibited similar influences. Parameter kG showed no influence on the modal properties.
- **Influence of Amount of Measured Modal Properties**: To estimate parameters K_t and EI, it was insufficient to use only one bending mode in both directions.
- **Influence of Measurement Uncertainties**: Parameter K_r showed limited sensitivity to the uncertainties in the measurements, with values obtained between 0.40 and 0.47, with a target value of 0.40. In contrast, parameters K_t and EI exhibited significant sensitivity to uncertainties. The values of parameter K_t ranged between 6.41 and 10.0, with a target value of 9.0. For EI, the values ranged between 10.29 and 12.28, with a target value of 10.5.
- **Estimating Structural Properties**: Despite incorporating additional features compared to the model used in the study by Moretti et al. [1], the discrete Timoshenko beam model was not sufficient for accurately describing the third bending mode. This was indicated by a MAC value of 0.87.
  However, when only the lowest two bending modes were used to fit the model, the estimates of parameter $K_{r,y}$ showed low uncertainty, with a Coefficient of Variation of 6.91%. This was an improvement compared to the study conducted by Moretti et al. [1], which obtained a Coefficient of Variation of 46.9% with a uniform Euler-Bernoulli beam model. Based on these results, it could be stated that estimating the structural parameters of the New Orleans using vibration-based model updating was effective.

6

The office tower the Delftse Poort

To evaluate the effectiveness of vibration-based model updating in estimating the structural properties of a shear-dominant high-rise building, which has irregular stiffness across its heights, using a discrete Timoshenko beam model, this chapter applied the technique to the office tower the Delftse Poort in Rotterdam.

Section 6.1 explains the studies conducted on the office tower the Delftse Poort. Section 6.2 shows the results of these studies. Finally, the main findings of the research are summarized in Section 6.3.

6.1. Studies

The research consists of six studies. The first five studies examine how different settings and uncertainties in the model updating process, using a discrete Timoshenko beam model, influence the results obtained. The final study applies model updating to estimate the structural properties of the Delfste Poort, using the discrete Timoshenko beam model and the measured modal properties. The details of the studies are explained in Sections 6.1.1 to 6.1.6. Appendix D describes how the measured modal properties were obtained.

6.1.1. Influence Features Model

The first study is about the features of the Timoshenko beam model. The model incorporates shear deformations and accounts for irregular stiffness across its height. To assess the influence of these features in the beam model, the natural frequencies and mode shapes obtained with this model are compared with those obtained using the uniform Timoshenko beam and discrete Euler-Bernoulli beam model.

The discrete Timoshenko beam model used in the studies to approximate the dynamic behavior of the Delftse Poort is created using the Finite Element (FE) model of the building. This FE model, shown in Figure 6.1, is divided into four segments. The segements were defined based on the floorplans. The floors within a segment have the same floorplan in the FE model. Therefore, a four-segment Timoshenko beam model is used to represent the FE model in Figure 6.1.

The initial structural property values applied in this beam model are shown in Table 6.1. These values were obtained using the information of the FE model and the floor plans. Details of the approach with which the structural property values were obtained are described in Appendix C. The second moment of area around the x-axis is denoted as I_x. The rotational stiffness of the foundation around the x-axis is denoted as $K_{r,x}$. The translation stiffness of the foundation in x-direction is denoted as $K_{t,x}$.

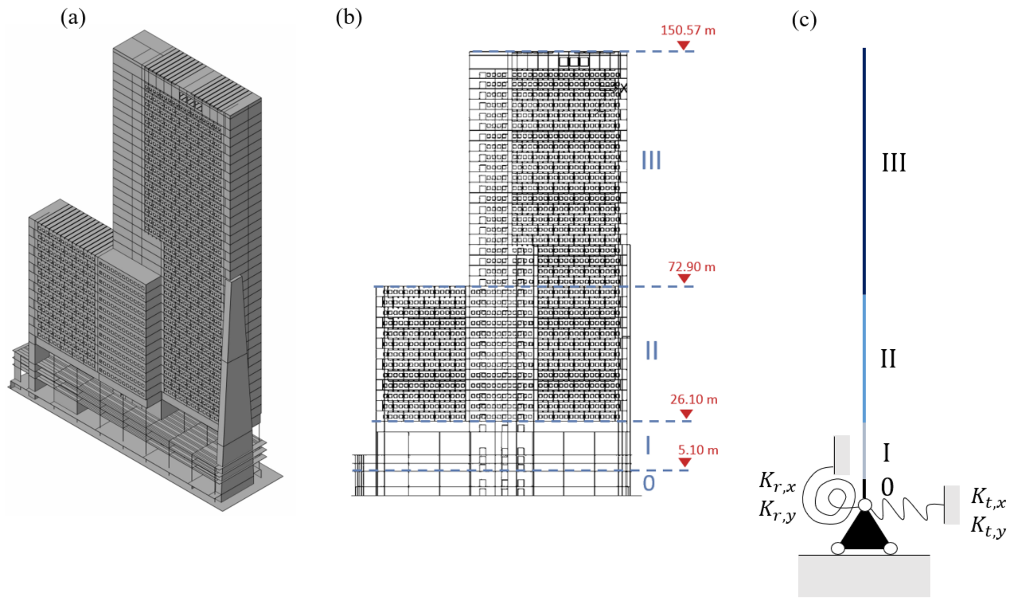

Figure 6.1: Pictures of (a) the FE model of the Delftse Poort, (b) the 4 segments defined for the Delftse Poort model, and (c) the four-segment Timoshenko beam model.

Segment	**0**	**I**	**II**	**III**
L [m]	3.15	22.65	46.8	81.27
A[m^2]	1320	1320	1320	825
I_x [m^4]	4312	4312	4744	1402
I_y [m^4]	34350	34350	35105	10685
ρ [kg/m^3]	5071	552	426	371
E[N/m^2]	11e9	11e9	22e9	33e9
v [-]	0.2	0.2	0.2	0.2
G [N/m^2]	4.6e9	4.6e9	9.2e9	1.38e9
k [-]	0.85	0.85	0.85	0.85
Boundary				
$K_{r,x}$ [GNm/rad]	6486	$K_{t,x}$ [GN/m]	9.22	
$K_{r,y}$ [GNm/rad]	50371	$K_{t,y}$ [GN/m]	9.22	

Table 6.1: The structural property values obtained with the floor plans and FE model of the Delftse Poort.

Obtaining these bending stiffnesses of the segments was challenging. Two approaches were applied to determine the bending stiffness EI_s (with s being x or y) per segment. The first approach calculated the bending stiffness per segment EI_s using the elastic modulus E applied in the FE model and the second moment of area I, calculated using the term $\frac{1}{12}bh^3$ and the Steiner rule:

$$EI_s = E\left(\frac{1}{12}bh^3 + bd^2\right) \tag{6.1}$$

The second approach considered the segments of the FE model. The FE model of the segment was clamped at the bottom, and a force F was applied at the top, resulting in a relative displacement Δ. The bending stiffness EI_s was then determined using this displacement Δ and Equation 6.2:

$$EI_s = \frac{FL^3}{3\Delta} \tag{6.2}$$

More information about the applied approaches can be found in Appendix C. The values obtained from each approach are shown in Table 6.2. Additionally, the displacements at the top, using either a four-segment beam model with these bending stiffnesses or the FE model, are presented, when a force of 10,000 kN is applied and the base of the model is fixed.

	$\mathbf{EI}_y$ [$\mathbf{Nm}^2$]		$\mathbf{EI}_x$ [$\mathbf{Nm}^2$]		FEM	
	Steiner	**FE segment**	**Steiner**	**FE segment**	**X**	**Y**
0	$614e12$	$1.04e12$	$40e12$	$0.26e12$		
I	$614e12$	$6.46e12$	$40e12$	$5.53e12$		
II	$1255e12$	$170.84e12$	$88e12$	$48.81e12$		
III	$573e12$	$255.56e12$	$39e12$	$41.60e12$		
Δu [mm]	16	-	236	-	26	199

Table 6.2: The bending stiffnesses and the total displacements determined with the different approaches.

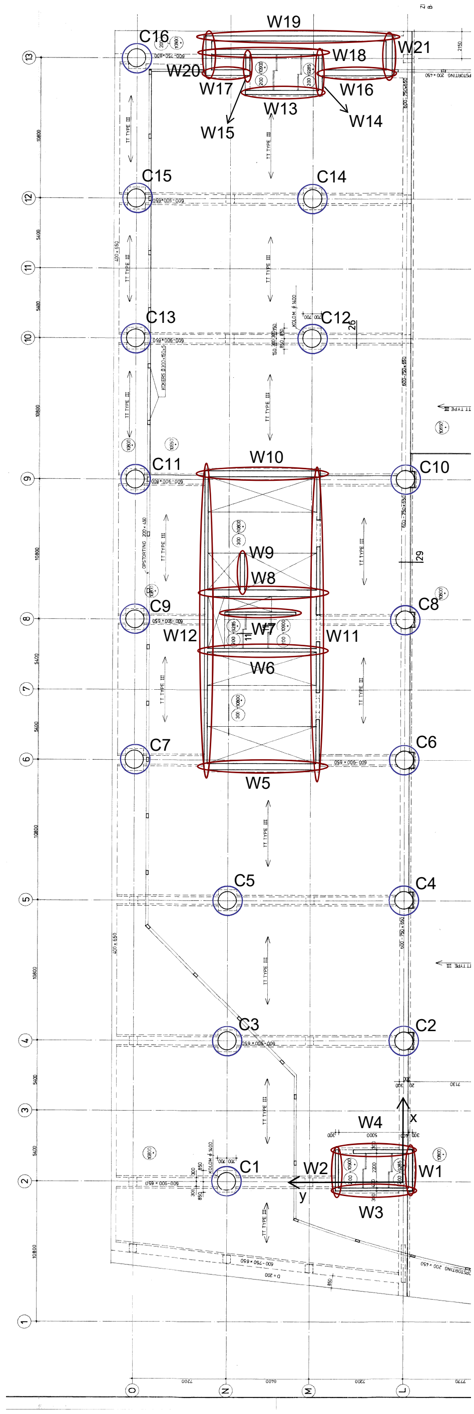

Figure 6.2: The floor plan of segment 0.

With the FE segment approach, as shown in Table 6.2, the bending stiffnesses in both x- and y-direction vary significantly. The top segment, which represent the smallest part of the building, shows the largest bending stiffness value. These differences are not observed with the bending stiffnesses obtained using the Steiner approach. The displacements obtained with the Steiner approach show similarities with the displacements obtained using the FEM model.

The FE segment approach fails to account for the intermediate relationships between the different segments. For example, segments 0 and I have column structures, which exhibit weaker behavior when considered individually than when integrated into the overall system, due the additional mass from other segments. The floor plan of segment 0 is given in Figure 6.2. The E-moduli values of the different segments in the FE model may also contribute to the differences, as the top segment has a E-modulus value of 33×10^9 N/m^2, while the bottom segment has a E-modulus value of 11×10^9 N/m^2). However, it is not expected that this caused the factor of 100 difference between the segment stiffnesses.

Due to the differences in the segment stiffnesses obtained using the FE segment approach, the segment stiffnesses obtained using the Steiner approach are applied, with a correction factor included to account for the displacement differences between the FE model and four-segment beam using these segment stiffnesses. Since the FE model is weaker in the x-direction, the segment stiffnesses EI_y are multiplied by $\frac{16}{26}$. Since the FE model is stiffer in the y-direction, the segment stiffnesses EI_x are multiplied by $\frac{236}{199}$.

It is assumed that the ratios of the initial structural properties between the segments are well estimated with the Steiner approach. By maintaining these ratios, only one set of parameters needs to be updated, thereby reducing the total amount of parameters that needs to be updated.

The uniform Timoshenko beam model and the discrete Euler-Bernoulli beam model used in this study are shown in Figure 6.3. For the discrete Euler-Bernoulli beam, the same initial structural properties are applied as those of the four-segment Timoshenko beam model. For the uniform Timoshenko beam model, the volume weighted averages of these values are used.

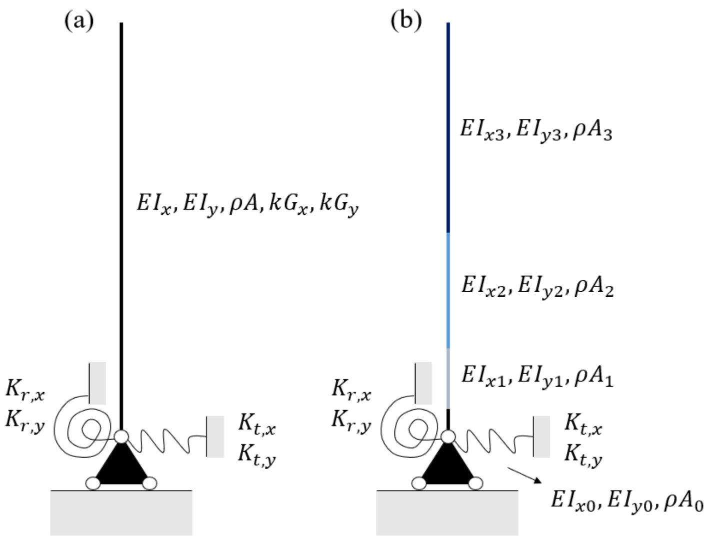

Figure 6.3: Pictures of (a) the uniform Timoshenko beam model and (b) the discrete Euler-Bernoulli beam model.

6.1.2. Influence Parameters

The second study investigate the influence of the input parameters of the model. With model updating, a decision must be made about which parameters to update. To understand how each input parameter of the discrete Timoshenko beam model influences the modal properties of the model (i.e., natural frequencies and mode shapes), a sensitivity study is conducted where the values of each input parameter are varied within a specified range.

The range over which each input parameter is varied corresponds to their initial structural property value in Table 6.1, scaled by a factor of ten. Parameters A and ρ, representing the cross-section and density, respectively, are excluded from this study, as these parameters do not address model uncertainties.

6.1.3. Influence Optimization Algorithm

In the third study, the influence of the optimization algorithm on model updating with a discrete Timoshenko beam model is investigated through a convergence study. The study uses three different algorithms on the same case: Sequential Least Squares Quadratic Programming (SLSQP), Differential Evolution (DE), and Particle Swarm Optimization (PSO). For each algorithm, different amounts of candidate solutions are considered. The updating parameters and the value ranges considered in this study are chosen based on the study "Influence Input Parameters" in Section 6.1.2.

To conduct this study, measured modal properties of a building with known structural properties are required. Since such data is not available, as the structural properties of the Delftse Poort are not known, a numerical case is created using the structural property values from Table 6.1. Random scalars are applied to the initial values in Tables 6.1 in both x- and y-direction, creating another model with known structural and modal properties. The random scalars applied are listed in Table 6.3. The modal properties of this new model serve as the measured modal properties for the study. The amount of modal properties applied aligns the amount of measured modal properties of the Delftse Poort: one bending mode in the x-direction and two bending modes in the y-direction. The modal displacements considered correspond to the measured heights of the Delftse Poort (-3.0 m, 69.30 m, 108.90 m and 144.51 m).

It should be noted that the random scaling factors are applied exclusively to the parameters selected for updating. Otherwise, it would be impossible to completely match the model output with the measured modal properties after model updating.

	K_r	K_t	E	I	ρ	kG
Scalar (-)	0.4	9.0	7.0	1.5	2.0	0.2

Table 6.3: The random scalar applied on the initial structural properties values of Table 6.2.

6.1.4. Influence Amount of Measured Modal Properties

In the fourth study, the influence of the amount of measured modal properties on model updating with the discrete Timoshenko beam model is examined. A convergence study is conducted that considers an increasing number of modes on the same case:

- The first bending mode in the x- and y-direction
- The first bending mode in the x- and the first two bending modes in y-direction

Different amounts of candidate solutions are considered for each amount of modes. The updating parameters and the value ranges considered in this study are chosen based on the study "Influence Input Parameters" in Section 6.1.2. The optimization algorithm is chosen based on the study "Influence Optimization Algorithm" in Section 6.1.3. Since this study also requires measured modal properties of a building with known structural properties, the numerical case from the study "Influence Optimization Algorithm" is also applied here.

6.1.5. Influence Measurements Uncertainties

In the fifth study, the impact of errors in the measured modal properties (i.e., the measured natural frequencies and mode shapes) on model updating with the discrete Timoshenko beam model is investigated. The sensitivity study introduces random errors into the measured modal properties to understand how these uncertainties affect the results obtained. The errors applied to the natural frequencies range up to 10%, and the errors applied to the mode shapes range up to 20%. An example of an error distributions is illustrated in Figure 6.4 for the first natural frequency and mode shape.

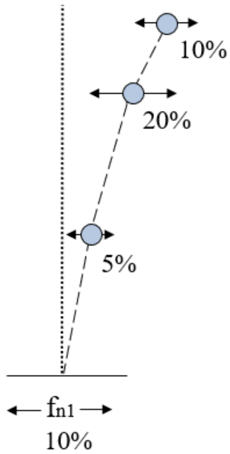

Figure 6.4: A visualization of the random errors applied on the first natural frequency and mode shape.

The updating parameters and the value ranges considered in this study are chosen based on the study ”Influence Input Parameters” in Section 6.1.2. The optimization algorithm is chosen based on the study ”Influence Optimization Algorithm” in Section 6.1.3. The amount of measured properties from Table 6.4 used to fit the model and the amount of candidate solution considered are based on the study ”Influence of the Amount of Measured Properties” in Section 6.1.4. Since this study also requires measured modal properties of a building with known structural properties, the numerical case from the study ”Influence Optimization Algorithm” is applied here as well.

6.1.6. Estimating Structural Properties

The last study applies model updating on the Delftse Poort to estimate its structural properties using the discrete Timoshenko beam model and the measured modal properties.

Mode	**1**	**2**	**3**
Dominant direction	Y	X	Y
Natural frequency [Hz]	0.403	0.861	1.624
Modal Displacement			
Height 1 (−3.30 m)	0.01	−0.06	0.01
Height 2 (69.30 m)	0.33	−0.49	0.49
Height 3 (108.90 m)	0.66	−0.76	0.12
Height 4 (144.51 m)	0.99	−1.00	−0.56

Table 6.4: The measured natural frequencies and modal displacements of the office tower the Delfste Poort.

The measured modal properties of the Delftse Poort are indicated in Table 6.4. To determine these modal properties, acceleration sensors were installed on the 1st, 19th, 30th, and 40th floors. The positions and orientations of the sensors are illustrated in Figure 6.5. Details of the data analysis applied to obtain the measured modal properties are described in Appendix D.

The updating parameters and the value ranges considered for these updating parameters are based on the study ”Influence Parameters” in Section 6.1.2. The algorithm applied in the model updating process is determined using the study ”Influence Optimization Algorithm” in Section 6.1.3. The amount of measured properties from Table 6.4 used to fit the model and the amount of candidate solution considered are based on the study ”Influence of the Amount of Measured Properties” in Section 6.1.4.

The model updating in this study follows the approach described by Moretti et al. [1] and consists of two parts. In the first part, model updating is carried out using the specified settings above. However, unlike the method used by Moretti et al. [1], which retains the lowest quantile of the candidate solutions (i.e., the quantile that best fit the measured modal properties after updating), this study

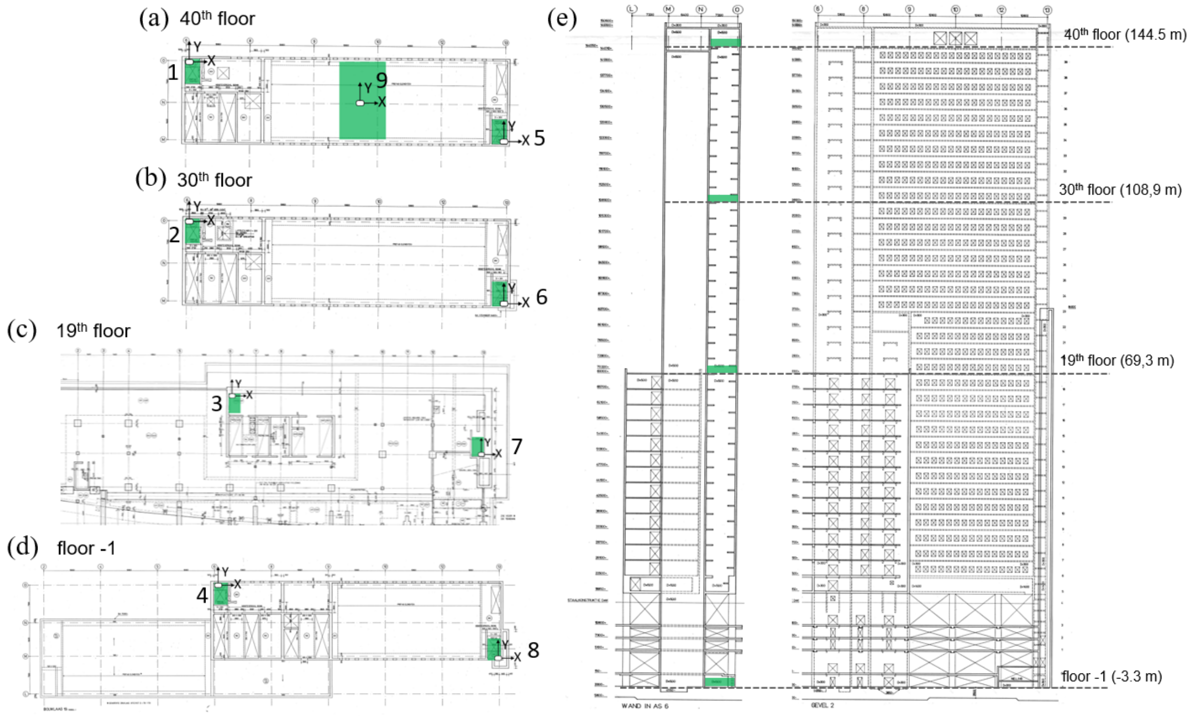

Figure 6.5: Pictures of (e) the Delfste Poort and the positions of the acceleration sensors (green squares) on (d) floor 1, on (c) the 19th floor, (b) the 30th floor and (a) the 40th floor. The arrows indicate the directions of the acceleration sensors.

retains the lowest octile of the candidate solutions. This modification is done because the discrete Timoshenko beam model is expected to be more complex than the uniform Euler-Bernoulli beam model, resulting in a more complex solution space for identifying the global minimum. New structural property value ranges for the updating parameters are defined based on the minimum and maximum values of these retained solutions. Using these newly refined ranges, another selection and model updating process is carried out (the second part) in the same manner as the first part. The reasons behind certain choices made by Moretti et al. [1] in this procedure are unclear. This approach was chosen for the Delfste Poort to stay consistent with the research conducted on the New Orleans in Chapter 5.

6.2. Results

In this section, the results of the studies are presented. Sections 6.2.1 to 6.2.5 present the results of the studies where the influences of certain settings and uncertainties in model updating with the discrete Timoshenko beam were investigated. Section 6.2.6 shows the results of applying model updating to estimate the structural properties of the Delftse Poort.

6.2.1. Influence Model

To begin, the results of the first study are presented. The natural frequencies obtained with the different beam models are shown in Tables 6.5 and 6.6. Their mode shapes are illustrated in Figure 6.6. The modal properties of the Finite Element (FE) model and the measured modal properties are also shown as reference.

According to Table 6.5, the discrete Timoshenko beam model is closer to the measured natural frequencies than the FE model. This is unexpected, as the FE model incorporates more realistic boundary conditions and better captures complex geometries compared to the discrete Timoshenko model. Moreover, the discrete Timoshenko beam model is created using the FE model. Therefore, it is unlikely that the discrete Timoshenko beam model provides a more accurate approximation of the high-rise building.

The uniform Timoshenko beam model generally matches the measured natural frequencies less accurately than the discrete Timoshenko beam model. This is expected, as a uniform Timoshenko beam model is not well-suited for representing buildings with irregular stiffness. A discrete beam model is needed for accurate representation. In the y-direction, the discrete Euler-Bernoulli beam model shows closer resembles to the measured natural frequencies than the discrete Timoshenko beam model. This suggests that excluding shear contributions may be more accurate for this direction.

Figure 6.6 shows that the modes shapes of the beam models and the FE model are comparable to the measured mode shapes. However, a notable difference is observed in the first bending mode in the x-direction at lower heights, where the FE model displays more displacement compared to the beam models. The FE model shows a kink. A thorough analysis of the FE model could provide valuable insights into the obtained results.

Measured [Hz]		**FEM (Δ_{measured} %) [Hz]**		**Discrete Timoshenko beam model (Δ_{measured} %) [Hz]**	
X	**Y**	**X**	**Y**	**X**	**Y**
0.86	0.40	0.62 (-27.9%)	0.28 (-30%)	0.76 (-11.6)	0.29 (-27.5)
	1.62		1.13 (-28.4%)		1.45 (-10.5%)

Table 6.5: The measured natural frequencies, the natural frequencies of the FE model and the natural frequencies of the discrete Timoshenko beam model.

Uniform Timoshenko beam model (Δ_{measured} %) [Hz]		**Discrete Euler-Bernoulli beam model (Δ_{measured} %) [Hz]**	
X	**Y**	**X**	**Y**
0.61 (-29.1)	0.23 (-42.5)	0.96 (+11.6)	0.37 (-7.5)
	1.37 (-15.4%)		1.61 (-0.6%)

Table 6.6: The natural frequencies obtained with the uniform Timoshenko and discrete Euler-Bernoulli beam models.

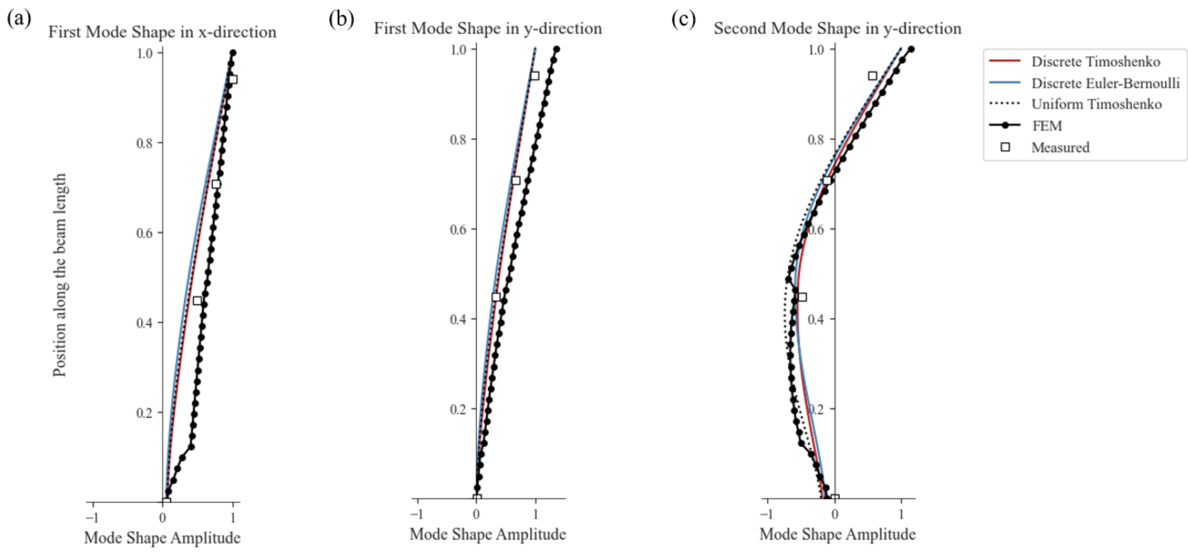

Figure 6.6: The measured and model mode shapes.

6.2.2. Influence Parameters

This section presents the results of second study, which changed the input parameter values of the discrete Timoshenko beam model to see their influence on the modal properties. The results of parameters K_r and K_t are shown in Figures 6.7 and 6.8. The results of parameters E and I are shown in Figures 6.10 and 6.9. The results of parameter kG is shown in Figure 6.11. The change in the mode shape is quantified using the MAC (Equation 2.2).

According to Figure 6.7, parameter K_r primarily has influence on the first natural frequency. It does not have influence on the mode shapes. Figure 6.8 shows that parameter K_t has influence on the first bending mode in the x-direction and the second bending mode in the y-direction, but have almost no influence on the first bending mode in the y-direction.

Parameters E and I have the same influence on the bending modes. As stated before with the New Orleans, these parameters often occupy similar positions within the model. Due to the similarity between parameters E and I, also for the Delftse Poort, it is chosen to consider the updating results of these parameters together as EI for the rest of the research.

Figure 6.11 shows that adjusting the value of the parameter kG has limited effect on the model output. It is not selected as an updating parameter. This is unexpected, as for the x-direction, the wide side of the building, a shear-dominant behavior was expected of the high-rise building. However, it should be noted that only the first bending mode is considered in x-direction. It is suspected that with higher bending modes, parameter kG shows more influence.

Based on the results, it is chosen to update parameters K_r, K_t, E, and I in the following studies of the research.

To summarize, it should be noted that parameter K_r primarily influence the first natural frequencies. Parameter K_t affects the first bending mode in the x-direction and the second bending mode in the y-direction, but does not have influence on the first bending mode in the y-direction. Parameters E and I exhibit similar influences. Therefore, the results for these parameters will be considered together as EI for the rest of the research. Parameter kG is not selected, as the modal properties of the model are insensitive to changes in this parameter.

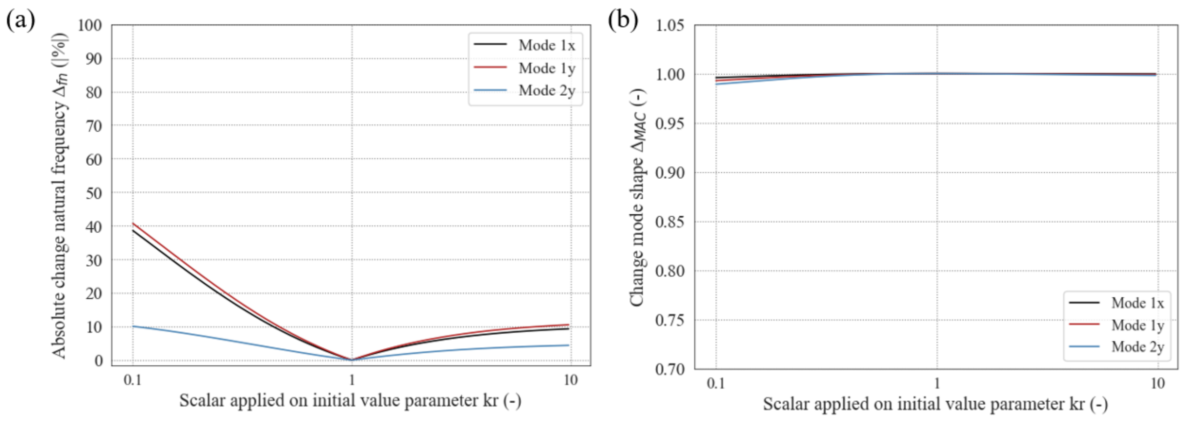

Figure 6.7: The influence of parameter K_r on the model output (a,b).

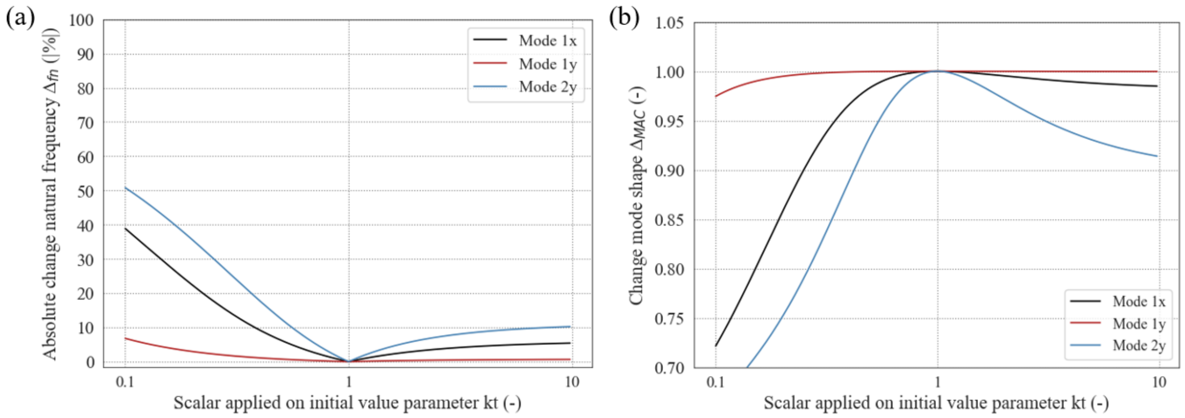

Figure 6.8: The influence of parameter K_t on the model output (a,b).

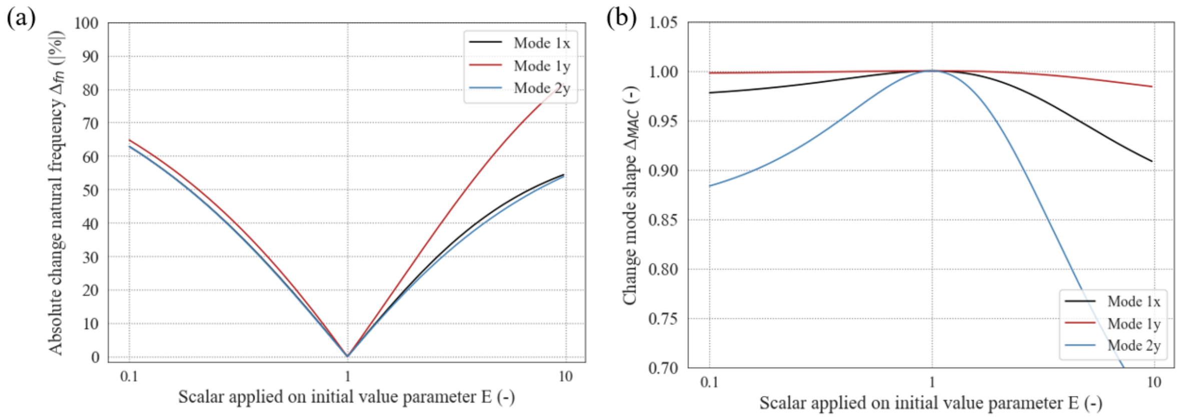

Figure 6.9: The influence of parameter E on the model output (a,b).

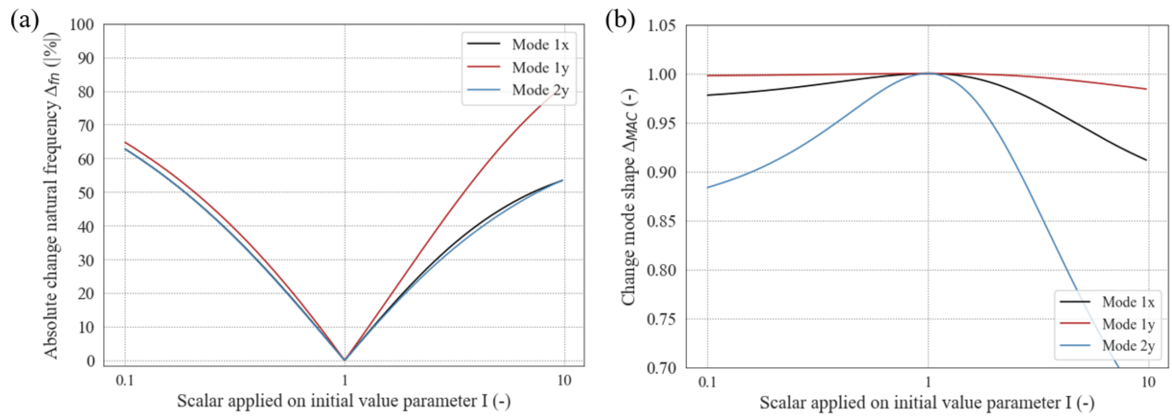

Figure 6.10: The influence of parameter I on the model output (a,b).

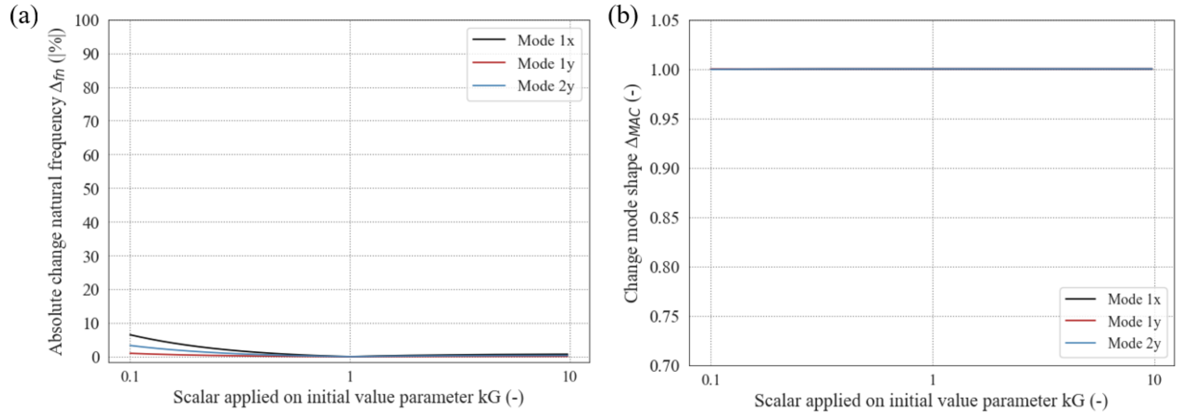

Figure 6.11: The influence of parameter kG on the model output (a,b).

6.2.3. Influence Optimization Algorithm

In this section, the results of the third study are shown, which applied different optimization algorithms on the same numerical case. The updating parameters were K_r, K_t, E and I, which were updated at the same time.

To create the numerical case, random scalars were applied to the initial values of the chosen updating parameters, indicated in Table 6.1, to generate another model with known structural properties and mode shapes. The scalars applied are indicated in Table 6.7. The modal properties of this new model, indicated in Table 6.8, were selected as measured modal properties for this study.

	K_r	K_t	E	I
Scalar (-)	0.4	9.0	7.0	1.5

Table 6.7: The random scalar applied on the initial structural properties values of Table 6.1.

Mode	**1**	**2**	**3**
Dominant direction	Y	X	Y
Natural frequency [Hz]	0.38	1.02	4.00
Modal Displacement			
Height 1 (−3.0 m)	0.00	0.02	−0.16
Height 2 (69.30 m)	0.45	0.47	−0.161
Height 2 (108.90 m)	0.74	0.74	−0.02
Height 3 (144.51 m)	1.00	1.00	1.00

Table 6.8: The modal properties of the numerical case.

Different amounts of candidate solutions were considered, and for each amount, the candidate solution that best fitted the measured modal properties after model updating is shown. To maintain the compactness of the results, the results are presented as ratios relative to their initial values in Table 6.1. For example, a value of 0.8 indicates that after the model updating, the new value of the structural property is 80% of its initial value. Therefore, if the model updating was successful with the chosen algorithm, the results should match the scalars listed in Table 6.7. The results of parameters K_r and K_t are shown in Figures 6.12 and 6.13, respectively. The results of parameters E and I are combined and shown as EI in Figure 6.14.

Figure 5.15 shows that K_r is well estimated by all algorithms. However, the optimization algorithms PSO and DE have difficulty with estimating parameter K_t. Increasing the amount of candidate solutions does not improve the estimates. This suggest that more candidate solutions need to be considered to sufficiently explore the solution space.

The SLSQP algorithm estimates parameter K_t well after 200 candidate solutions, however, small deviations are observed. The algorithm has more difficulty with estimating this parameter. Small deviation can be expected, as the model updating process stops once a certain convergence threshold is reached, which may occur earlier for some initiations than others. Therefore, this algorithm is selected for the following studies.

Figure 6.14 shows that EI is well estimated by all optimization algorithms.

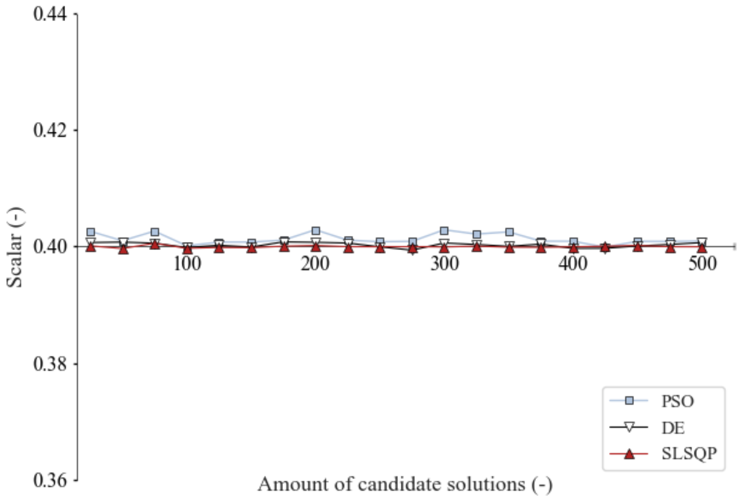

Figure 6.12: The influence of the different algorithms on the model updating result of parameter K_r.

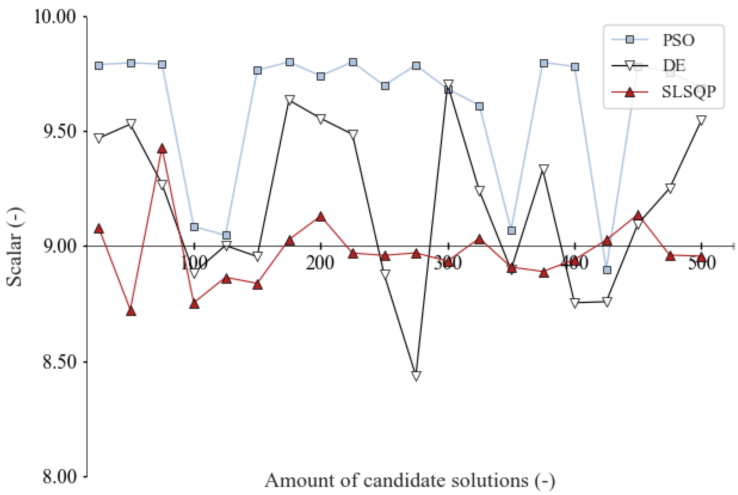

Figure 6.13: The influence of the different algorithms on the model updating result of parameter K_t.

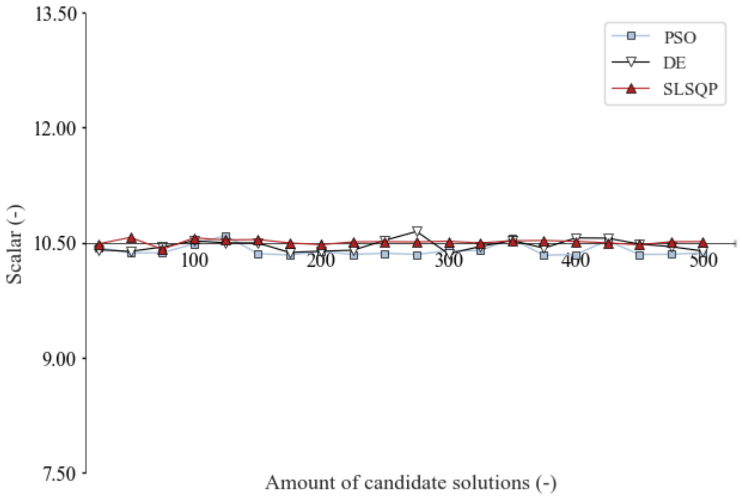

Figure 6.14: The influence of the different algorithms on the model updating result of the product EI.

6.2.4. Influence Amount of Measured Modal Properties

This section presents the model updating results of the fourth study, which applied different amount of measured modal properties on the same numerical case. The parameters chosen to update in this study were K_r, K_t, E and I, which were updated at the same time. The SLSQP was chosen as optimization algorithm.

This study used the same numerical case described in the study ”Influence Optimization Algorithm” in Section 6.2.3. Different amounts of candidate solutions were considered, and for each amount, the candidate solution that best fitted the measured modal properties after updating is shown. The results per parameter are shown as ratios relative to their initial values. Therefore, if the model updating was successful with the amount of measured modal properties, the results should match the scalars listed in Table 6.7. The results of the parameters K_r and K_t are shown in Figures 6.15 and 6.16, respectively. The results of the parameters E and I are shown together as EI in Figure 6.17.

Figure 6.15 shows that it is possible to estimate parameter K_r considering only the lowest bending mode in both x-and y-direction. This also in line with the influences obtained in Figure 6.7 in Section 6.2.2, as it showed that this parameter primarily influence the first natural frequency.

Figures 6.16 and 6.17 show, however, that it is insufficient to estimate K_t and EI considering only the lowest bending mode in the x- and y-direction. According to Figure 6.8 in Section 6.2.2, parameter K_t shows only influence on the first bending mode in the x-direction and the second bending mode in the y-direction. Excluding the second bending mode in the y-direction results in insufficient information to estimate this parameter.

However, according to 6.9 and 6.10, parameters E and I show sensitivity to the first bending mode. It is suspected that the first bending mode alone does not provide enough information to estimate these parameters. Higher bending modes, which are more sensitive to changes in bending stiffness, are needed to estimate these parameters.

To estimate parameter K_t, a minimum of 200 candidate solutions is required. Based on the results, it is chosen to consider all the measured modal properties listed in Table 6.18 in the study ”Estimating Structural Properties”, using 400 candidate solutions. For the study ”Influence of Measurement Uncertainties”, 200 candidate solutions are used.

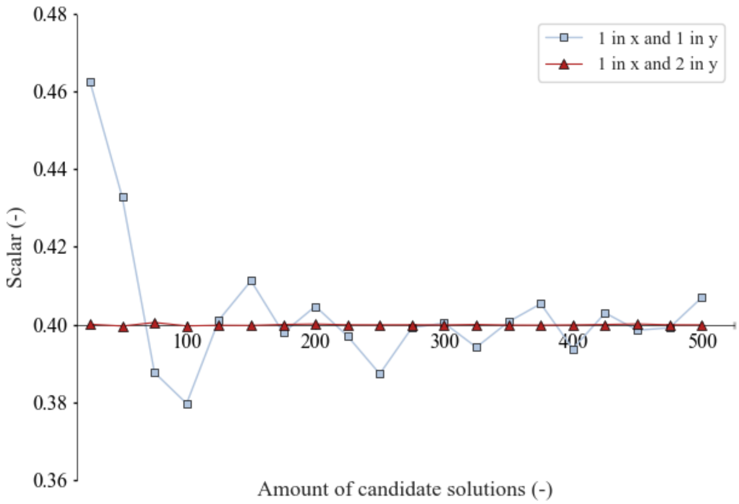

Figure 6.15: The influence of the amount of modal information on the model updating result of parameter K_r.

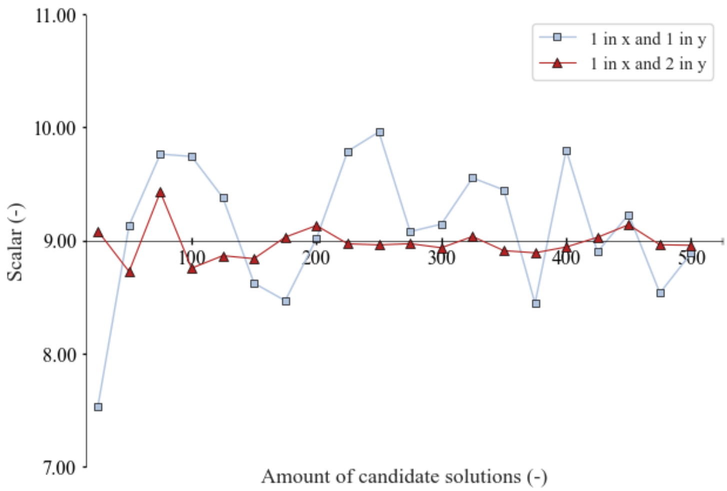

Figure 6.16: The influence of the amount of modal information on the model updating result of parameter K_t.

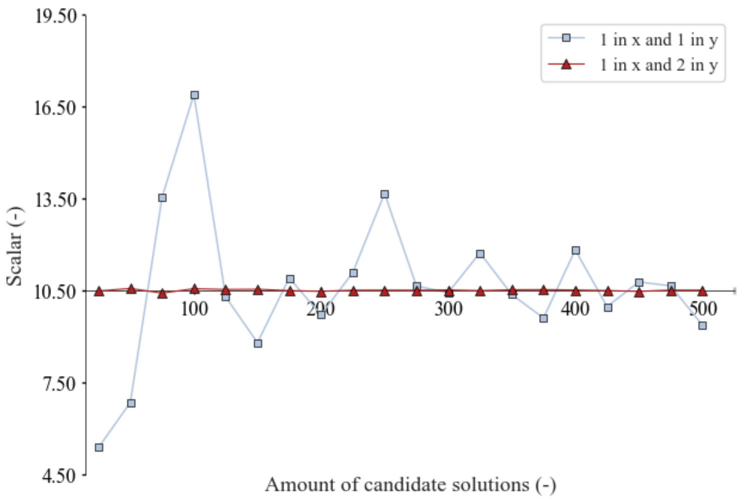

Figure 6.17: The influence of the amount of modal information on the model updating result of EI.

6.2.5. Measurement Uncertainties

This section shows the results of the fifth study, which applied errors to the measured modal properties to see their influence on the results obtained. The parameters updated in this study were K_r, K_t, E and I. These were updated at the same time, using the SLSQP optimization algorithm. The study used the same numerical case described in the study "Influence Optimization Algorithm" in Section 6.2.3. Two hundred candidate solution were considered, but only the candidate solution that best fitted the measured modal properties is shown. As the procedure of adding random errors, conducting model updating, and retaining the best-fitted solution was done ten times, ten solutions are shown in total.

The results for each parameter are presented as ratios relative to their initial values. Therefore, if the model updating was insensitive to the added errors in the measured modal properties, the results should match the scalars listed in Table 6.7. The results are shown in Figure 6.2.5. The values of the different parameters for each solution are connected by grey lines. The red solution indicates the scalar values of Table 6.7.

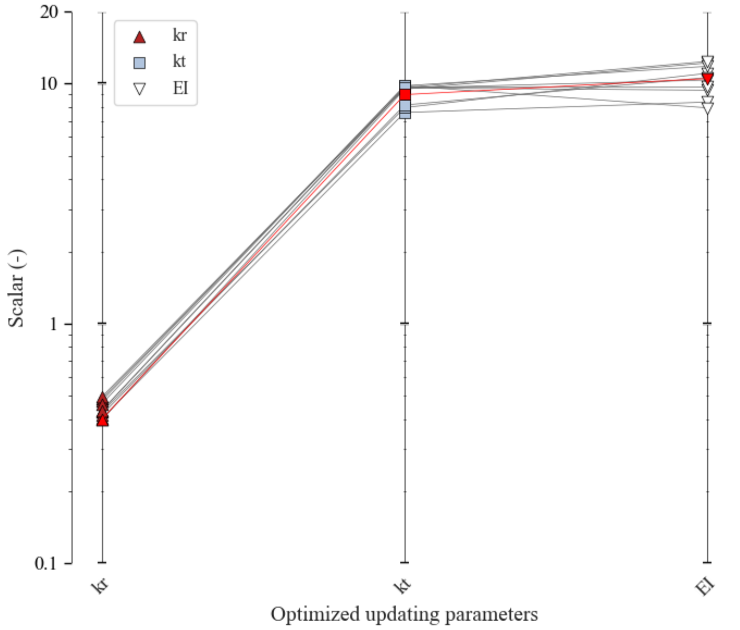

Figure 6.18: The influence of measurement uncertainties on the model updating results.

Parameter K_r shows limited sensitivity to the uncertainties in the measured model properties. The values of this parameter are between 0.39 and 0.49. K_t and EI, however, show significant sensitivity to the uncertainties. The values of parameter K_t range between 7.58 and 10.00. The values of EI range between 7.93 and 12.38.

6.2.6. Estimating Structural Properties

To end, the estimated structural properties values of the Delftse Poort are shown. The structural parameters updated in this study were K_r, K_t, E and I. The SLSQP algorithm was applied. Four hundred candidate solutions were considered, and the lowest octile of these solutions, i.e., the octile with the smallest objective function values, were retained.

Table 6.9 presents the estimated and measured natural frequencies and the Modal Assurance Criterion (MAC) values between the estimated and measured mode shapes. Case 1 indicates the results of model updating using the measured modal properties of Table 6.4.

The median natural frequency for Case 1 in the x-direction is close to the measured natural frequencies. However, in the y-direction, the median of the second natural frequency shows differences compared to the measured second natural frequency.

Mode [-]	Natural frequency [Hz]			MAC [-]
(dominant direction)	Design	Measured	Case 1 (Δ_{measured} %)	Measured vs. Case 1
1 (x)	0.620	0.861	0.861 (0%)	0.999
1 (y)	0.280	0.403	0.403 (0%)	0.999
2 (y)	1.130	1.624	1.803 (+11%)	0.910

Table 6.9: The measured and estimated natural frequencies and MAC values between the measured and estimated mode shapes.

Additionally, the MAC values indicate that the second bending mode shape in the y-direction is not accurately estimated. Therefore, the obtained structural properties values in y-direction are not considered.

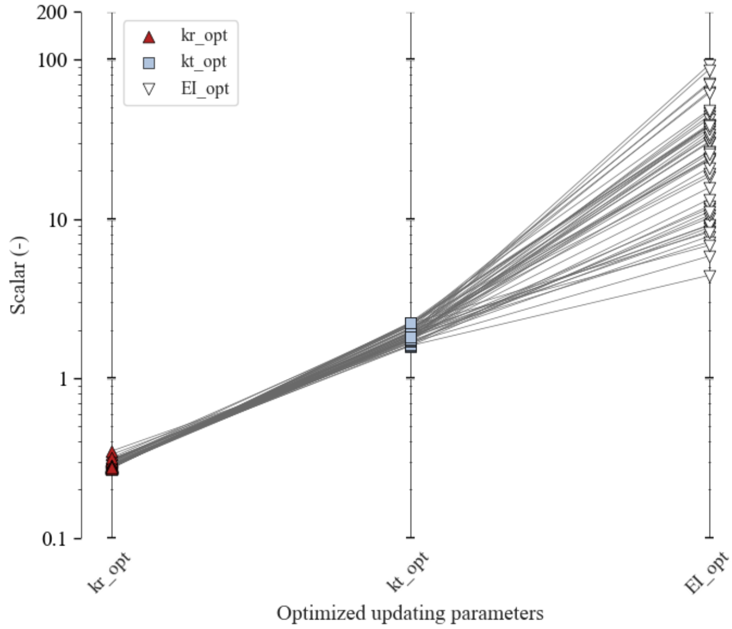

Figure 6.19: The estimated parameter values of Case 1 in x-direction.

Figure 6.19 shows the estimated structural property values in the x-direction. The values of the different parameters that belong together are connected with grey lines. For the estimates per parameter, the Coefficient of Variation (CV) is calculated. The CV shows the relative variability of the estimates:

$$\text{CV} = \left(\frac{\sigma}{\mu}\right) \times 100\% \tag{6.3}$$

where σ is the standard deviation and μ is the mean. A lower CV indicates less variability relative to the mean, suggesting more consistency in the estimates. A higher CV indicates greater dispersion in the estimates. The variability in the distribution of $K_{r,y}$ is low, indicated by a coefficient of variation (CV) of 5.58%. The distribution suggests that the value should be lower than the design value. The variability of the distribution of $K_{t,x}$ is also low, indicated by a coefficient of variation (CV) of 9.49%. The distribution suggests that the value should be higher than the design value.

However, the distribution of the parameter EI_y exhibit high uncertainty, as the estimates are scattered across the range. The coefficients of variation is 71.17%. Based on these results, it can be stated that estimating the structural parameters of the Delftse Poort using vibration-based model updating was not effective.

6.3. Main Findings

With the research, several studies were conducted to evaluate the influence of certain settings and uncertainties in model updating and to assess the effectiveness of the technique in estimating the structural properties of a shear-dominant building with irregular stiffness across its height. The main findings were:

- **Setup model:** Obtaining accurate values for bending stiffness using the FE model proved difficult. The FE approach, which used the relative displacement Δ of the FE segment to calculate the bending stiffness, did not account for the intermediate relationships between segments. Segments 0 and I had column structures, which exhibited weaker behavior when considered individually compared to when they were part of the overall system. This was due to the additional mass of the other segments. Therefore, considering these segments individually led to an inaccurate assessment of their bending stiffness. The FE segment approach resulted in bending stiffness values that differed by a factor of 100.
- **Influence of Parameters:** Parameter K_r showed primarily influence on the first natural frequencies. Parameter K_t showed influence on the first bending mode in the x-direction and the second bending mode in the y-direction, but did not show influence on the first bending mode in the y-direction. Parameters E and I exhibited similar influences. Parameter kG showed no influence on the modal output.
- **Influence of Amount of Modal Properties:** It was possible to estimate parameter K_r considering only the lowest bending mode in both x- and y-directions. However, this was insufficient to estimate K_t and EI.
- **Influence of Measurement Uncertainties:** Parameter K_r showed limited sensitivity to the uncertainties in the measurements, with values obtained between 0.39 and 0.49, closely aligning with the expected value of 0.40. In contrast, parameters K_t and EI exhibited significant sensitivity to uncertainties. The values of parameter K_t ranged between 7.58 and 10.00, with the expected value being 9.0. For EI, the values ranged between 7.93 and 12.38, while the expected value was 10.5.
- **Estimating Structural Properties:** The discrete Timoshenko beam model was not able to match the natural frequency of the measured second bending mode in the y-direction after updating. Additionally, the MAC values indicated that the second bending mode shape in the y-direction was not accurately estimated.
  Moreover, the distribution of the parameters EI_y exhibited high uncertainty, as the estimates were scattered across the range. The coefficient of variation was 71.17%. Based on these results, it can be stated that estimating the structural parameters of the Delftse Poort using vibration-based model updating was not effective.

Discussion

In this research, various studies were conducted on the New Orleans and the Delftse Poort to investigate the effectiveness of vibration-based model updating with a discrete Timoshenko beam model in estimating the structural properties of high-rise buildings. Their findings require explanation, which is provided in this chapter.

First, the setup of the models need to be discussed. The discrete Timoshenko beam models used to approximate the dynamic behavior of the New Orleans and the Delftse Poort were created using their Finite Element (FE) models. However, obtaining values for the bending stiffnesses using the FE model was challenging.

The Steiner approach, which calculated the bending stiffness per segment using the elastic modulus E and the second moment of inertia I (calculated with $\frac{1}{12}bh^3$ and the Steiner rule), assumed fully rigid connections between elements when determining I, making it not possible to accurately account for the contributions of elements that were not rigidly connected.

Moreover, the FE approach, which used the relative displacement Δ of the FE segment to calculate the bending stiffness per segment, also posed problems, as it did not account for the intermediate relationships between the segments. For both the New Orleans and the Delftse Poort, segments 0 and I had column structures, which exhibited weaker behavior when considered individually compared to when they were part of the overall system, due to the additional mass of the other segments. Therefore, considering these segments individually led to an inaccurate assessment of their bending stiffnesses. This was particularly evident in the case of the Delftse Poort, where the FE segment approach resulted in bending stiffness values differing by a factor of 100.

These difficulties in determining the bending stiffness values per segment pose a problem for model updating, as the ratios of the initial structural properties were maintained for both high-rise building applications. Due to the superstructure of the discrete Timoshenko beam model, a separate set of parameters needs to be defined for each segment. By maintaining the ratios of the initial structural property values, only one set of parameters needs to be updated, thereby reducing the total number of parameters that need to be updated. However, maintaining these ratios gives weight to the initial structural property values. These values must be correct. Otherwise, model updating is limited by the incorrect ratios and will not be able to accurately match the measured modal properties. Therefore, to obtain a more accurate estimation of the structural properties, it is crucial to obtain accurate values for the bending stiffness.

Despite these challenges with the bending stiffnesses, applying the discrete Timoshenko beam model for model updating to estimate the structural properties of the New Orleans has proven to be effective. When only the lowest two bending modes in both directions were used to fit the model, the estimates of parameter $K_{r,y}$ showed low uncertainty, with a Coefficient of Variation of 6.91%. This is an improvement compared to the study conducted by Moretti et al. [1], which obtained a Coefficient of Variation of 46.9% with the uniform Euler-Bernoulli beam model. Therefore, to obtain a more accurate estimation of parameter $K_{r,y}$, it is crucial to incorporate shear deformations into the model and account for irregular stiffness along the height. However, further investigation is needed to clarify whether the improved results of the structural property $K_{r,y}$ are primarily attributed to the discreteness of the beam, the added shear or a combination of both.

Despite incorporating additional features compared to the uniform Euler-Bernoulli beam model used in the study by Moretti et al. [1], the model was not sufficient for accurately describing the third bending mode.

This is indicated by a Modal Assurance Criterion (MAC) value of 0.87. This suggests that essential aspects are missing from the model that are crucial for matching the mode shape of the third bending mode.

Applying model updating with the discrete Timoshenko beam model to estimate the structural properties of the Delftse Poort was not effective. Parameter EI_y exhibits large uncertainty, which is indicated by a Coefficient of Variation of 71.17%. Several factors could contribute to this discrepancy. One possible cause might be that the value ranges considered for parameters E and I are too narrow. Errors in the initial values of these parameters could also be the cause. As previously mentioned, accurately estimating the values for bending stiffness was challenging. As the ratios of the initial structural properties values were maintained, the model updating may be constrained by incorrect ratios, which prevents it from accurately matching the measured modal properties. Moreover, only one bending mode in the x-direction was use to fit the discrete Timoshenko model. It is suspected that the first bending mode alone does not provide enough information to estimate parameters E and I. Higher bending modes, which are more sensitive to the bending stiffness, are needed, as they provide more comprehensive information.

However, the need for higher bending modes poses challenges for model updating, as higher bending modes are more likely to contain errors. For both the New Orleans and the Delftse Poort, K_t and EI show sensitivity to the measurement uncertainties. When errors in the natural frequency of up to 10% and in the mode shapes of up to 20% were applied, the values of K_t and EI for the New Orleans ranged from 6.41 to 10.0 and from 10.29 to 12.28, respectively. For the Delftse Poort, the values of K_t and EI ranged from 7.58 to 10.0 and from 7.93 to 12.38, respectively. The target values for K_t and EI for both cases were 9.0 and 10.5, respectively. Therefore, to achieve a more accurate estimation of these parameters, it is essential to minimize the errors of the measured modal properties to below these thresholds. This is not the case for parameter K_r. For both applications, parameter K_r did not exhibit significant sensitivities to measurement uncertainties. For the New Orleans, parameter K_r showed values between 0.40 and 0.47. For the Delftse Poort, parameter K_r showed values between 0.39 and 0.49. The target value of this parameter was 0.4.

Furthermore, for the Delftse Poort, the discrete Timoshenko beam model was unable to match the natural frequency of the measured second bending mode in the y-direction after updating. Additionally, the Modal Assurance Criterion (MAC) values indicated that the second bending mode shape in the y-direction was not accurately estimated. This result was unexpected, as the discrete Timoshenko beam model was able to successfully match the second measured bending mode for the New Orleans in both the x- and y-directions.

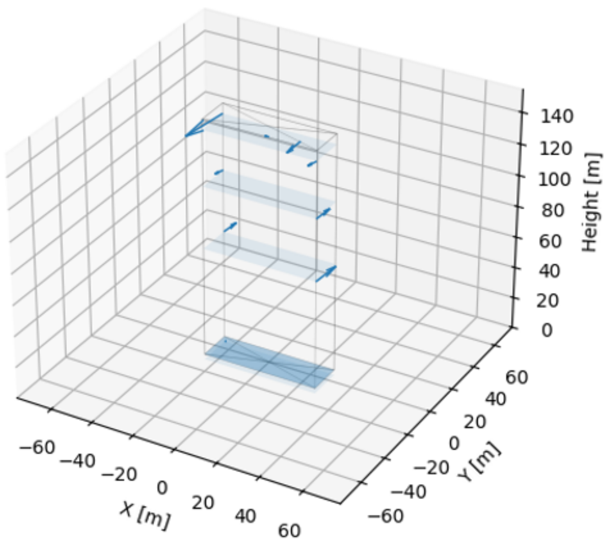

Figure 6.20: The mode shape of the second bending mode in y-direction of the Delftse Poort.

The measured mode shape is shown in 3D in Figure 6.20. The differences in the magnitude and direction of the arrows indicate twisting of the building. This twisting cannot be modeled with the discrete Timoshenko beam model, as it only can model pure bending modes. Furthermore, since the discrete Timoshenko beam model requires a single value per height, while multiple sensors were used to measure the displacements, the measurements had to be averaged. This averaging process may have led to a mode shape that deviates from the actual behavior, resulting in an inaccurate representation of reality. These issues pose a challenge for model updating with the Delftse Poort, as now only the first measured bending modes could be accurately applied, which provide alone insufficient information to estimate the parameters EI and K_t. Therefore, more pure bending modes are needed to obtain an accurate estimation of these parameters.

Conclusion

This research focused on applying vibration-based model updating to estimate the structural properties of high-rise buildings using a discrete Timoshenko beam model. It provided valuable insights into how the model affects the accuracy of estimating structural properties for high-rise buildings and expanded the use of vibration-based model updating to buildings that are shear-dominant in behavior and have irregular stiffness distributions across their height. The main objective of the research was to achieve a more accurate estimation of the structural properties of high-rise buildings using vibration-based model updating. Several key aspects were highlighted as essential to obtaining a more accurate estimation of these properties using model updating with a discrete Timoshenko beam model :

- **Discrete superstructure and shear:** For a more accurate estimation of parameter $K_{r,y}$, it is of importantce to incorporate shear deformations into the model and account for irregular stiffness along the height. For the New Orleans, when only the lowest two bending modes in both directions were used to fit the model, the estimates of parameter $K_{r,y}$ showed low uncertainty, with a Coefficient of Variation of 6.91%. This is an improvement compared to the study conducted by Moretti et al. [1], which obtained a Coefficient of Variation of 46.9% with the uniform Euler-Bernoulli beam model. Therefore, it is of importance to incorporate shear deformations into the model and account for irregular stiffness along the height.
- **Bending stiffness values:** To obtain a more accurate estimation of the structural properties, it is crucial to obtain accurate values of the bending stiffness. The discrete Timoshenko beam models used to approximate the dynamic behavior of the New Orleans and the Delftse Poort were created using their Finite Element (FE) models. However, obtaining values for the bending stiffnesses using the FE models proved challenging. The Steiner approach, which calculated the bending stiffness per segment using the elastic modulus E and the second moment of inertia I (calculated with $\frac{1}{12}bh^3$ and the Steiner rule), assumed fully rigid connections between elements when determining I, making it impossible to accurately account for the contributions of elements that were not rigidly connected. The FE approach, which used the relative displacement Δ of the FE segment to calculate the bending stiffness, did not account for the intermediate relationships between segments, leading to inaccurate estimation of the segments with column structures. These segments exhibit weaker behavior when considered individually compared to when they are part of the overall system due to the additional mass of the other segments.

  These challenges in determining bending stiffness values per segment posed a problem for model updating, as the initial structural property ratios were maintained for both high-rise buildings. Maintaining these ratios gave weight to the initial structural property values. These values must be correct. Otherwise, model updating is limited by the incorrect ratios and will not be able to accurately match the measured modal properties. Therefore, it is crucial to obtain accurate values of the bending stiffness.
- **High-quality measured modal properties:** To achieve a more accurate estimation of the parameters K_t and EI, it is of importance to obtain measured natural frequencies with errors below 10% and measured mode shapes with errors below 20%. For both the New Orleans and the Delftse Poort, K_t and EI showed sensitivity to measurement uncertainties. When errors in the measured natural frequencies of up to 10% and in the measured mode shapes of up to 20% were applied, the values of K_t and EI for the New Orleans ranged from 6.41 to 10.0 and from 10.29 to 12.28, respectively. For the Delftse Poort, the values of K_t and EI ranged from 7.58 to 10.0 and from 7.93 to 12.38, respectively. The target values for K_t and EI were for both cases 9.0 and 10.5, respectively. This posed challenges for model updating, as both parameters required higher bending modes for accurate estimation, which are more prone to errors. Therefore, it is essential to minimize the errors of the measured modal properties to below these thresholds.
- **Lack of pure bending modes:** To obtain a more accurate estimation of the parameters EI and K_t for the Delftse Poort, more measured pure bending modes are needed.

The discrete Timoshenko beam model was unable to match the natural frequency of the measured second bending mode in the y-direction after updating. Additionally, the Modal Assurance Criterion (MAC) values indicated that the second bending mode shape in the y-direction was not accurately estimated. The mode shape indicated twisting of the building, which could not be captured by the discrete Timoshenko beam model, as it is only capable of representing pure bending modes.

Furthermore, since the discrete Timoshenko beam model requires a single displacement value per height but multiple sensors were used to measure displacements, the measurements had to be averaged. This averaging process may have led to a mode shape that deviated from the actual behavior, resulting in an inaccurate representation of reality. These issues posed a challenge for model updating with the Delftse Poort, as now only the first measured bending modes in both direction could be accurately applied, which provided alone insufficient information to estimate the parameters EI and K_t. More pure bending modes are needed to obtain an accurate estimation of these parameters.

Based on these findings, several recommendations can be made to improve the accuracy and broaden the application of vibration-based model updating in estimating the structural properties of high-rise buildings. The first recommendation addresses the difficulties observed with the discrete superstructure. Maintaining the ratios gave weight to the initial structural property values, which needed to be correct for accurate estimations of the structural properties. It is recommended to allow some flexibility in these ratios, acknowledging that they might not be entirely accurate. Additionally, exploring model updating without keeping these ratios constant would be valuable. However, given the large number of parameters involved, this could potentially lead to an overdetermined problem, as independently adjusting the parameters of a discrete superstructure might result in multiple combinations that fit the measured modal properties equally well.

The second recommendation involves the improvement of the estimates of parameter $K_{r,y}$, indicated with a Coefficient of Variation of 6.91%. It remains uncertain whether the improved results of $K_{r,y}$ are primarily due to the discreteness of the beam, the added shear, or a combination of both. Therefore, it is recommended to perform model updating using both the uniform Timoshenko and the discrete Euler-Bernoulli beam model to compare the results obtained with those obtained with the discrete Timoshenko beam model.

The third recommendation involves torsion, a feature not currently accounted for in the discrete Timoshenko beam model. The discrete Timoshenko beam model was unable to accurately capture the second bending mode in the y-direction for the Delftse Poort, where twisting of the building was observed. This limitation suggests that torsion, which was not included in the current model, could be significant. Incorporating torsion into the model could address this issue by allowing the model to capture the twisting of the building more accurately. Moreover, including torsion in the model would enable the use of measured torsional modes, which could reveal new relationships between input parameters and the model output.

The last recommendation emphasizes the importance of obtaining measured data with minimal errors. It is advised to install additional sensors at various heights to reduce measurement errors. Increasing the number of sensors helps average out errors, leading to more accurate data.

References

[1] D. Moretti, A. J. Bronkhorst, and C. P. W. Geurts. "Identification of the structural properties of a high-rise building". In: (May 2023).

[2] E. Taciroglu, S.F. Ghahari, and F. Abazarsa. "Efficient model updating of a multi-story frame and its foundation stiffness from earthquake records using a timoshenko beam model". In: *Soil Dynamics and Earthquake Engineering* 92 (Jan. 2017), pp. 25–35.

[3] Dr. Garth N. Wells. *The Finite Element Method: An Introduction.* University of Cambridge and Delft University of technology, 2009.

[4] Olek C. Zienkiewicz, Robert L. Taylor, and J. Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals.* Elsevier, May 2005.

[5] J. E. Mottershead and M. I. Friswell. "Model Updating In Structural Dynamics: A Survey". In: *Journal of Sound and Vibration* 167.2 (Oct. 1993), pp. 347–375.

[6] Sergey E. Lyshevski. *Control Systems Theory with Engineering Applications.* Springer Science & Business Media, June 2001.

[7] M. I. Friswell and J. E. Mottershead. *Finite Element Model Updating in Structural Dynamics.* Ed. by G. M. L. Gladwell. Vol. 38. Solid Mechanics and its Applications. Springer Netherlands, 1995.

[8] H. G. Natke, ed. *Identification of Vibrating Structures.* Springer Vienna, 1982.

[9] B. A. H. Abbas and J. Thomas. "The second frequency spectrum of timoshenko beams". In: *Journal of Sound and Vibration* 51.1 (Mar. 1977), pp. 123–137.

[10] Seyedali Mirjalili. *Genetic Algorithm | SpringerLink.* Vol. 780. Springer, June 2018.

[11] N. A. Z. Abdullah et al. "A review on model updating in structural dynamics". In: *IOP Conference Series: Materials Science and Engineering* 100.1 (Nov. 2015).

[12] A. J. Bronkhorst, D. Moretti, and C. P. W. Geurts. "Identification of the Dynamic Properties of the Residential Tower New Orleans". In: *Experimental Vibration Analysis for Civil Engineering Structures.* Ed. by Zhishen Wu et al. Vol. 224. Springer International Publishing, 2023, pp. 469–479.

[13] S. F. Ghahari et al. "Blind identification of the Millikan Library from earthquake data considering soil–structure interaction". In: *Structural Control and Health Monitoring* 23.4 (2016), pp. 684–706.

[14] R. W. TRAILL-NASH and A. R. COLLAR. "THE EFFECTS OF SHEAR FLEXIBILITY AND ROTATORY INERTIA ON THE BENDING VIBRATIONS OF BEAMS". In: *The Quarterly Journal of Mechanics and Applied Mathematics* 6.2 (Mar. 1953), pp. 186–222.

[15] G. R. Bhashyam and G. Prathap. "The second frequency spectrum of Timoshenko beams". In: *Journal of Sound and Vibration* 76.3 (June 1981), pp. 407–420.

[16] M. Levinson and D. W. Cooke. "On the two frequency spectra of Timoshenko beams". In: *Journal of Sound and Vibration* 84.3 (Oct. 1982), pp. 319–326.

[17] A. J. Bronkhorst and C. P. W. Geurts. "Long-term vibration and wind load monitoring on a high-rise building". In: *Proceedings ISMA2020, Leuven, Belgium* (2020).

[18] S.P. Timoshenko. "LXVI. *On the correction for shear of the differential equation for transverse vibrations of prismatic bars*". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.245 (May 1921), pp. 744–746.

[19] Antonio Cazzani, Flavio Stochino, and Emilio Turco. "On the whole spectrum of Timoshenko beams. Part I: a theoretical revisitation". In: *Zeitschrift für angewandte Mathematik und Physik* 67.2 (Apr. 2016), p. 24.

[20] Karl F. Graff. *Wave motion in Elastic Solids*. Oxford: Dover publications (New York), 1991.

[21] J.M.J. Spijkers, A.W.C.M. Vrouwenvelder, and E.C. Klaver. *Structural Dynamics CT4140 - Part 1- Structural Vibrations*. Delft University of Technology - Faculty of Civil Engineering and Geosciences, Jan. 2005.

[22] Angelo Simone. "An Introduction to the Analysis of Slender Structures". In: *Delft University of Technology, Faculty of Civil Engineering and Geosciences - Structural Mechanics Section - Computational Mechanics Group* (Sept. 2011).

[23] Rune Brincker, Lingmi Zhang, and Palle Andersen. "Modal identification of output-only systems using frequency domain decomposition". In: *Smart Materials and Structures* 10.3 (June 2001), p. 441.

[24] J. R. Wu and Q. S. Li. "Finite element model updating for a high-rise structure based on ambient vibration measurements". In: *Engineering Structures* 26.7 (June 2004), pp. 979–990.

[25] *ChatGPT*.

[26] *Vertalen met DeepL Translate*. nl.

A

Timoshenko beam model

The Timoshenko beam, which was first introduced by Timoshenko in 1921 [18], is a model used to describe the dynamic behavior of a beam. The beam model, illustrated in Figure A.1, accounts for both bending and shear deformation [19], and is utilized in many papers published after 1921. Despite the amount of papers published after 1921, however, discussion regarding its exact definition remain, especially due to the existence of a so-called 'second spectrum of modes'.

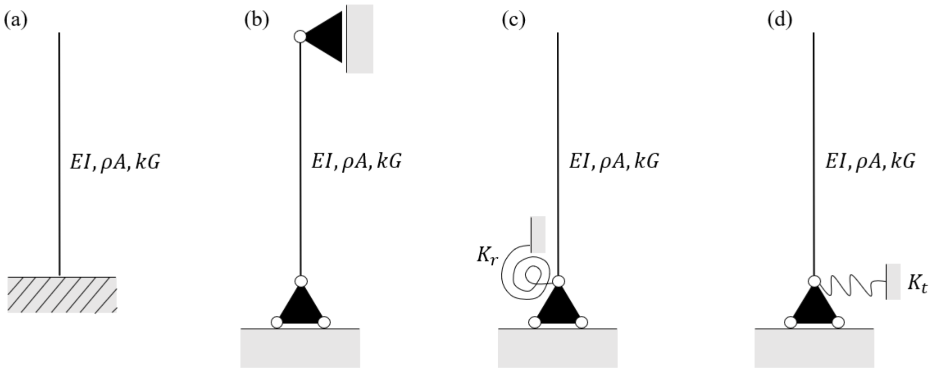

Figure A.1: The Timoshenko beam model (1D) with various boundary conditions: (a.) a clamped beam, (b.) a simply supported beam, (c.) a beam supported by a rotational spring and (d.) a beam supported by a translational spring.

Traill-Nash and Collar were the first to indicate the presence of a second spectrum of modes [14]. Their findings caused various responses: Abbas and Thomas argued that, aside from a simply supported beam, no second spectrum of modes exists, stating that earlier conclusions about its existence were misinterpretations [9]. Bhashyam and Prathap provided proof of its existence for various boundary conditions, and developed a methodology to categorize the different modes [15]. Levinson and Cooke claimed that, even for the simply supported Timoshenko beam, only one spectrum of modes exists [16].

The ongoing debate may explain why many papers published since 1921 do not provide a complete solution and often fail to address the potential existence of a second spectrum of modes. To address this gap and to provide a more detailed explanation of the model, this appendix presents the complete derivation of a piece-wise Timoshenko beam. It begins with the governing equations of dynamics, followed by an eigenvalue analysis of the beam model. Additionally, it outlines the boundary and continuity conditions applied. The appendix concludes with a numerical example to verify the procedure and illustrate how the phenomenon of the second spectrum of modes appears.

A.1. The governing equations of dynamics

To start the derivation of the Timoshenko beam, a reference system is needed. The applied reference system is illustrated in Figure A.2.

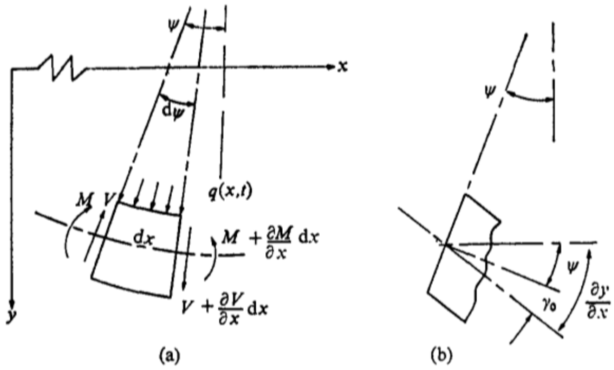

Figure A.2: The sign convention of the Timoshenko beam of the publication 'Wave motion in Elastic Solids' of Karl F. Graff [20]. This sign convention is used in this Appendix.

In the figure, M represents the bending moment, V represents the shear force, $q(x,t)$ represents the distributed force, $\frac{\partial y}{\partial x}$ represents the slope of the centroidal axis, γ_0 represents the shear strain at centroidal axis, and ψ represents the rotation of the plane. The curvature of the plane, denoted as $\frac{1}{R}$, equals to the first derivative of ψ. The kinematic equations are written as [20]:

$$\begin{aligned} \gamma_0 &= \frac{\partial y}{\partial x} - \psi \\ \frac{1}{R} &= -\frac{d\psi}{dx} \end{aligned} \tag{A.1}$$

To relate the kinematic equations with the applied loads, constitutive relations are applied. These equations are written as [20]:

$$\begin{aligned} Q &= G\int \gamma dA = kG\gamma_0 A \\ M &= \int y\sigma(y)dA = E\frac{1}{R}\int y^2 dA = -EI\frac{\partial\psi}{\partial x} \end{aligned} \tag{A.2}$$

With these relations, γ_0 is used for the entire cross-section. This, however, may result in differences compared to the situation when the variable shear stress is integrated across the section. To correct this difference, a coefficient k, known as the Timoshenko shear coefficient, is introduced. This coefficient depends on the shape of the cross-section [20].

To connect the applied loads with the internal forces and moments, equilibrium equations are applied. The equilibrium equations in the ψ- and y- directions are written as [20]:

$$\begin{aligned} \sum T &= dM - Vdx = \rho I dx\frac{\partial^2\psi}{\partial t^2} \\ \sum F_y &= dV + q(x,t)dx = ma_y = \rho A dx\frac{\partial^2 y}{\partial t^2} \end{aligned} \tag{A.3}$$

Substituting the kinematic and constitutive relations in these equilibrium equations gives the governing equations for the dynamics of a Timoshenko beam model [20]:

$$\begin{aligned} EI\frac{\partial^2\psi(x,t)}{\partial x^2} + kGA(\frac{\partial y(x,t)}{\partial x} - \psi(x,t)) &= \rho I\frac{\partial^2\psi(x,t)}{\partial t^2} \\ kGA(\frac{\partial^2 y(x,t)}{\partial x^2} - \frac{\partial\psi(x,t)}{\partial x}) + q(x,t) &= \rho A\frac{\partial^2 y(x,t)}{\partial t^2} \end{aligned} \tag{A.4}$$

The equations are a system of second-order partial differential equations (PDEs).

A.2. Eigenvalue problem

To find the eigenvalue problem, the system of second-order partial differential equations is reduced to one fourth-order partial differential equation using the method separation of variables. To start this procedure, the governing equations of dynamics are given as a set of homogeneous equations [21]:

$$\begin{aligned} EI\frac{\partial^2\psi(x,t)}{\partial x^2} + kGA(\frac{\partial y(x,t)}{\partial x} - \psi(x,t)) - \rho I\frac{\partial^2\psi(x,t)}{\partial t^2} &= 0 \\ kGA(\frac{\partial^2 y(x,t)}{\partial x^2} - \frac{\partial\psi(x,t)}{\partial x}) - \rho A\frac{\partial^2 y(x,t)}{\partial t^2} &= 0 \end{aligned} \tag{A.5}$$

In these equations, the term $q(x,t)$ is set to zero, indicating a non-loaded beam. To decouple the set of equations, the variables (x,t) and $\phi(x,t)$ are decomposed into the product of two unknown functions x and t [21]:

$$\begin{aligned} y(x,t) &= Y(x)T(t) = Y(x)\sin(\omega t+\phi) \\ \psi(x,t) &= \Psi(x)T(t) = \Psi(x)\sin(\omega t+\phi) \end{aligned} \tag{A.6}$$

The unknown function $T(t)$ is assumed to have a sinusoidal shape $\sin(\omega t+\phi)$. Substituting equations A.6 into equations A.5 yields [21]:

$$\begin{aligned} (\frac{d^2\Psi(x)}{dx^2} + \frac{kGA}{EI}(\frac{dY(x)}{dx} - \Psi(x)) + \frac{\rho I\omega^2}{EI}\Psi(x))\sin(\omega t+\phi) &= 0 \\ (\frac{d^2Y(x)}{dx^2} - \frac{d\Psi(x)}{dx} + \frac{\rho A\omega^2}{kGA}Y(x))\sin(\omega t+\phi) &= 0 \end{aligned} \tag{A.7}$$

The time function $\sin(\omega t+\phi)$ is not equal to zero at all times. Therefore, it is required that the term within brackets is independent of time [21]:

$$\begin{aligned} \frac{d^2\Psi(x)}{dx^2} + \frac{kGA}{EI}(\frac{dY(x)}{dx} - \Psi(x)) + \frac{\rho I\omega^2}{EI}\Psi(x) &= 0 \\ \frac{d^2Y(x)}{dx^2} - \frac{d\Psi(x)}{dx} + \frac{\rho A\omega^2}{kGA}Y(x) &= 0 \end{aligned} \tag{A.8}$$

Substituting these equations into each other gives the fourth-order partial differential equation of the Timoshenko beam model [21], known as the eigenvalue problem:

$$EI\frac{d^4Y(x)}{dx^4} + \frac{\rho\omega^2}{kGA}(EIA + kGAI)\frac{d^2Y(x)}{dx^2} + \frac{\rho\omega^2}{kGA}(\rho IA\omega^2 - kGA^2)Y(x) = 0 \tag{A.9}$$

A.3. Eigenvalue analysis

To solve the eigenvalue problem indicated above, an eigenvalue analysis is performed. For convenience, dimensionless parameters are defined to rewrite the equation [2]:

$$\begin{aligned} s^2 &= \frac{EI}{kGAL^2} \\ b^2 &= \frac{\rho A\omega^2 L^4}{EI} \\ R^2 &= \frac{r^2}{L^2} \end{aligned} \tag{A.10}$$

Rewriting equation A.9 with these parameters give A.11:

$$\frac{d^4\tilde{Y}(\tilde{x})}{d\tilde{x}^4} + (b^2s^2 + b^2R^2)\frac{d^2\tilde{Y}(\tilde{x})}{d\tilde{x}^2} + (b^4s^2R^2 - b^2)\tilde{Y}(\tilde{x}) \tag{A.11}$$

To solve this eigenvalue problem, a general solution is assumed in the form of an exponential function $\tilde{Y}(\tilde{x}) = \exp\lambda\tilde{x}$. Implementing this assumption results in the characteristic equation of the system [2]:

$$\lambda^4 + (b^2s^2 + b^2R^2)\lambda^2 + (b^4s^2R^2 - b^2) = 0 \tag{A.12}$$

Solving this characteristic equation gives four solutions, referred to as the eigensolutions of the problem [2]:

$$\begin{aligned} \lambda_{1,2} &= \pm\sqrt{\frac{-(b^2s^2+b^2R^2)-\sqrt{(b^2s^2+b^2R^2)^2-4(b^4s^2R^2-b^2)}}{2}} \\ \lambda_{3,4} &= \pm\sqrt{\frac{-(b^2s^2+b^2R^2)+\sqrt{(b^2s^2+b^2R^2)^2-4(b^4s^2R^2-b^2)}}{2}} \end{aligned} \tag{A.13}$$

The roots $\lambda_{1,2}$ are always imaginary. The roots $\lambda_{3,4}$, however, can be real or imaginary, depending on the frequency value. The change from real to imaginary occurs at a certain point in the frequency spectrum, known as the *transition frequency* ω_c. The transition frequency is equal to $\omega_c = \sqrt{\frac{kGA}{\rho I}}$. Consequently, the frequency spectrum can be seperated into two parts, with a special case at the transition frequency ω_c [2]:

Case 1: $\omega < \sqrt{\frac{kGA}{\rho I}}$
In this case, the roots $\lambda_{3,4}$ are real, giving the real eigenfunctions as [2]:

$$\begin{aligned} \tilde{Y}(\tilde{x}) &= A_1\sin(p\tilde{x}) + A_2\cos(p\tilde{x}) + A_3\sinh(q\tilde{x}) + A_4\cosh(q\tilde{x}) \\ \tilde{\Phi}(\tilde{x}) &= B_1\sin(p\tilde{x}) + B_2\cos(p\tilde{x}) + B_3\sinh(q\tilde{x}) + B_4\cosh(q\tilde{x}) \end{aligned} \tag{A.14}$$

where the p is equal to $|\mathrm{Im}(\lambda_1)|$ and the q is equal to λ_3. The coefficients B_1, B_2, B_3 and B_4 are related to the coefficients A_1, A_2, A_3 and A_4 through the equations of A.8 [2]:

$$\begin{aligned} B_1 &= \left(\frac{b^2s^2}{p} - p\right)A_2 \\ B_2 &= \left(-\frac{b^2s^2}{p} + p\right)A_1 \\ B_3 &= \left(\frac{b^2s^2}{q} + q\right)A_4 \\ B_4 &= \left(\frac{b^2s^2}{q} + q\right)A_3 \end{aligned} \tag{A.15}$$

It should be noted that $\tilde{Y}(\tilde{x}) = Y(x)/L$, $\frac{d\tilde{Y}(\tilde{x})}{dx} = \frac{dY(x)}{dx}$, $\frac{d^2\tilde{Y}(\tilde{x})}{dx^2}/L = \frac{dY(x)}{dx}$ and $\frac{d\tilde{Y}(\tilde{x})}{dx}/L = \frac{dY(x)}{dx}$ [2].

Case 2: $\omega = \sqrt{\frac{kGA}{\rho I}}$
In this case, the roots $\lambda_{3,4}$ are equal to zero, giving the real eigenfunctions A.16 as [2]:

$$\begin{aligned} \tilde{Y}(\tilde{x}) &= C_1\sin(p\tilde{x}) + C_2\cos(p\tilde{x}) + C_3\tilde{x} + C_4 \\ \tilde{\Phi}(\tilde{x}) &= D_1\sin(p\tilde{x}) + D_2\cos(p\tilde{x}) + b^2s^2D_3\tilde{x} + D_4 \end{aligned} \tag{A.16}$$

where the p is equal to $|\mathrm{Im}(\lambda_1)|$ and $C_3 = 0$. The coefficients D_1, D_2, D_3 and D_4 are related to C_1, C_2, C_3 and C_4 through the equations of A.8 [2]:

$$\begin{aligned} D_1 &= \left(\frac{b^2s^2}{p} - p\right)C_2 \\ D_2 &= \left(-\frac{b^2s^2}{p} + p\right)C_1 \\ D_3 &= C_4 \end{aligned} \tag{A.17}$$

Case 3: $\omega > \sqrt{\frac{kGA}{\rho I}}$
In this case, the roots $\lambda_{3,4}$ are imaginary, giving the real eigenfunctions A.18 as [2]:

$$\begin{aligned} \tilde{Y}(\tilde{x}) &= E_1\sin(p\tilde{x}) + E_2\cos(p\tilde{x}) + E_3\sin(q\tilde{x}) + E_4\cos(q\tilde{x}) \\ \tilde{\Phi}(\tilde{x}) &= F_1\sin(p\tilde{x}) + F_2\cos(p\tilde{x}) + F_3\sin(q\tilde{x}) + F_4\cos(q\tilde{x}) \end{aligned} \tag{A.18}$$

where the p is equal to $|\mathrm{Im}(\lambda_1)|$ and the q is equal to $|\mathrm{Im}(\lambda_3)|$. The coefficients F_1, F_2, F_3 and F_4 are related to E_1, E_2, E_3 and E_4 through the equations of A.8 [2]:

$$
\begin{aligned}
F_1 &= \left(\frac{b^2 s^2}{p} - p\right) E_2 \\
F_2 &= \left(-\frac{b^2 s^2}{p} + p\right) E_1 \\
F_3 &= \left(\frac{b^2 s^2}{p} - p\right) E_4 \\
F_4 &= \left(-\frac{b^2 s^2}{p} + p\right) E_3
\end{aligned}
\tag{A.19}
$$

A.4. Boundary and continuity conditions

To fit the eigenfunctions to the actual problem, boundary conditions are applied [22]. These conditions, listed in Table A.1, give information about the geometric character (*kinematic boundary conditions*) or the forces (*dynamic boundary conditions*) of the beam.

Boundary	**Conditions**	
Hinged	$\tilde{Y}(\tilde{x}) = 0$	$\frac{d\tilde{\Psi}(\tilde{x})}{d\tilde{x}} = 0$
Fixed	$\tilde{Y}(\tilde{x}) = 0$	$\tilde{\Psi}(\tilde{x}) = 0$
Free	$\frac{d\tilde{\Psi}(\tilde{x})}{d\tilde{x}} = 0$	$\tilde{\Psi}(\tilde{x}) - \frac{d\tilde{Y}(\tilde{x})}{d\tilde{x}}$
Rotational	$\tilde{Y}(\tilde{x}) = 0$	$\frac{d\tilde{\Psi}(\tilde{x})}{d\tilde{x}} - \frac{k_r}{\frac{EI}{L}}\tilde{\Psi}(\tilde{x}) = 0$
Translational	$\tilde{\Psi}(\tilde{x}) - \frac{d\tilde{Y}(\tilde{x})}{d\tilde{x}} + \frac{k_t}{\frac{kGA}{L}}\tilde{Y}(\tilde{x}) = 0$	$\tilde{\Psi}(\tilde{x}) = 0$
Rotational + Translational	$\tilde{\Psi}(\tilde{x}) - \frac{d\tilde{Y}(\tilde{x})}{d\tilde{x}} + \frac{k_t}{\frac{kGA}{L}}\tilde{Y}(\tilde{x}) = 0$	$\frac{d\tilde{\Psi}(\tilde{x})}{d\tilde{x}} - \frac{k_r}{\frac{EI}{L}}\tilde{\Psi}(\tilde{x}) = 0$

Table A.1: The boundary conditions of a Timoshenko beam [21, 2, 19].

With a piece-wise beam model, it is also necessary to consider continuity conditions. These conditions, listed in Table A.2, explain the connection between the intermediate parts of the beam [22]. They represent the continuity of displacement, forces, and the slope of the beam between the parts.

Boundary	**Conditions**
Continuity	$\tilde{Y}_{left}(\tilde{x}) = \tilde{Y}_{right}(\tilde{x})$ $\frac{d\tilde{Y}_{left}(\tilde{x})}{d\tilde{x}} = \frac{d\tilde{Y}_{right}(\tilde{x})}{d\tilde{x}}$ $\frac{d\tilde{\Psi}_{left}(\tilde{x})}{d\tilde{x}} = \frac{EI_{right}}{EI_{left}}\frac{d\tilde{\Psi}_{right}(\tilde{x})}{d\tilde{x}}$ $\tilde{\Psi}_{left}(\tilde{x}) - \frac{d\tilde{Y}_{left}(\tilde{x})}{d\tilde{x}} = \frac{kGA_{right}}{kGA_{left}}(\tilde{\Psi}_{right}(\tilde{x}) - \frac{d\tilde{Y}_{right}(\tilde{x})}{d\tilde{x}})$

Table A.2: The continuity conditions of a piece-wise Timoshenko beam [22].

A.5. The eigenvalues and the eigenmodes

To determine the unknown coefficients of the eigenfunctions and thereby obtain the eigenmodes of the beam, the eigenfunctions are incorporated into the boundary and continuity conditions of the system, resulting in a system of homogeneous algebraic equations (in matrix formulation) [2]:

$$\mathbf{Av} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = 0$$

where $\mathbf{v}$ is the vector with the unknown coefficients and $\mathbf{A}$ is the coefficient matrix. In mathematical terms, this comes down to determining the eigenvectors. For the piece-wise Timoshenko beam, for each discrete part, a local coefficient matrix is created, which is then incorporated into a global coefficient matrix $\mathbf{A}$:

$$\mathbf{Av} = 0$$

$$\mathbf{A} = \begin{bmatrix}
a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & & & & & & & & \\
a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & & & & & & & & \\
a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} & a_{3,8} & & & & \\
a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} & & & & \\
a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} & a_{5,7} & a_{5,8} & & & & \\
a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} & a_{6,7} & a_{6,8} & & & & \\
 & & & & a_{7,5} & a_{7,6} & a_{7,7} & a_{7,8} & & & & \\
 & & & & a_{8,5} & a_{8,6} & a_{8,7} & a_{8,8} & & & & \\
 & & & & a_{9,5} & a_{9,6} & a_{9,7} & a_{9,8} & & & & \\
 & & & & a_{10,5} & a_{10,6} & a_{10,7} & a_{10,8} & & & & \\
 & & & & & & \ddots & & \ddots & & & \\
 & & & & & & & & a_{n-5,n-3} & a_{n-5,n-2} & a_{n-5,n-1} & a_{n-5,n} \\
 & & & & & & & & a_{n-4,n-3} & a_{n-4,n-2} & a_{n-4,n-1} & a_{n-4,n} \\
 & & & & & & & & a_{n-3,n-3} & a_{n-3,n-2} & a_{n-3,n-1} & a_{n-3,n} \\
 & & & & & & & & a_{n-2,n-3} & a_{n-2,n-2} & a_{n-2,n-1} & a_{n-2,n} \\
 & & & & & & & & a_{n-1,n-3} & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\
 & & & & & & & & a_{n,n-3} & a_{n,n-2} & a_{n,n-1} & a_{n,n}
\end{bmatrix}$$

$$\mathbf{v} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ \vdots \\ \\ c_{n-6} \\ c_{n-5} \\ c_{n-4} \\ c_{n-3} \\ c_{n-1} \\ c_n \end{bmatrix}$$

The index n refers to the amount of unknown coefficients. The set of algebraic equations possesses a non-trivial solution if the determinant of this coefficient matrix equals zero [2]:

$$\det(\mathbf{A}) = 0$$

This equation is known as the frequency equation. The zero points of the frequency equation represent the omega values of the system. At these points, the eigenvectors are determined which contain the relations between the coefficients using $\mathbf{Av} = 0$. This procedure concludes the derivation of the Timoshenko beam model [2].

A.6. Numerical example

To verify the procedure and to show the phenomenon of the second spectrum of modes, a numerical example is applied. The parameters of this examples are shown in Table A.3. The chosen example is an uniform simply supported beam [19]. A two-piece Timoshenko beam model is used to replicate the uniform beam.

Segment	I / II
L [m]	2
B [m]	0.1
I [m^4]	$\frac{1}{120000}$
E [N/m^2]	$260e9$
v [-]	0.3
G [N/m^2]	$100e9$
ρ [kg/m^3]	8000

Table A.3: The parameters (with units) of a uniform simply supported beam [19]. To verify the code, the results of this beam are reproduced using a two-piece Timoshenko beam. With that beam, each discrete element is given the same parameters as the uniform beam to obtain the same results as for the uniform beam.

Case 1: $\omega < \sqrt{\frac{kGA}{\rho I}}$

In the first part of the spectrum, 25 eigenvalues are obtained, shown in table A.4. The obtained eigenvalues with the two-piece Timsohenko beam model show no differences with the article, confirming the accuracy of the Python code for the first part of the spectrum. In Figures A.3 and A.4, the first five eigenmodes of the spectrum are shown.

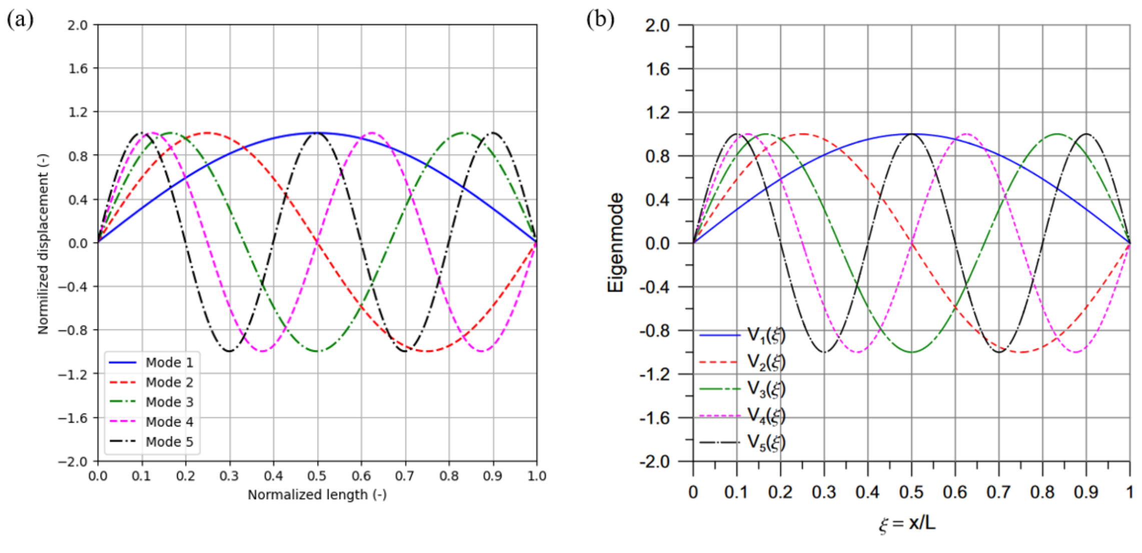

Figure A.3: A illustration of the first five mode shapes : (a.) shows the results of $Y(x)$ obtained with the two-piece Timoshenko beam model and (b.) shows the results of the article [19]. The functions V_1, V_2, V_3, V_4 and V_5 are defined in article [19].

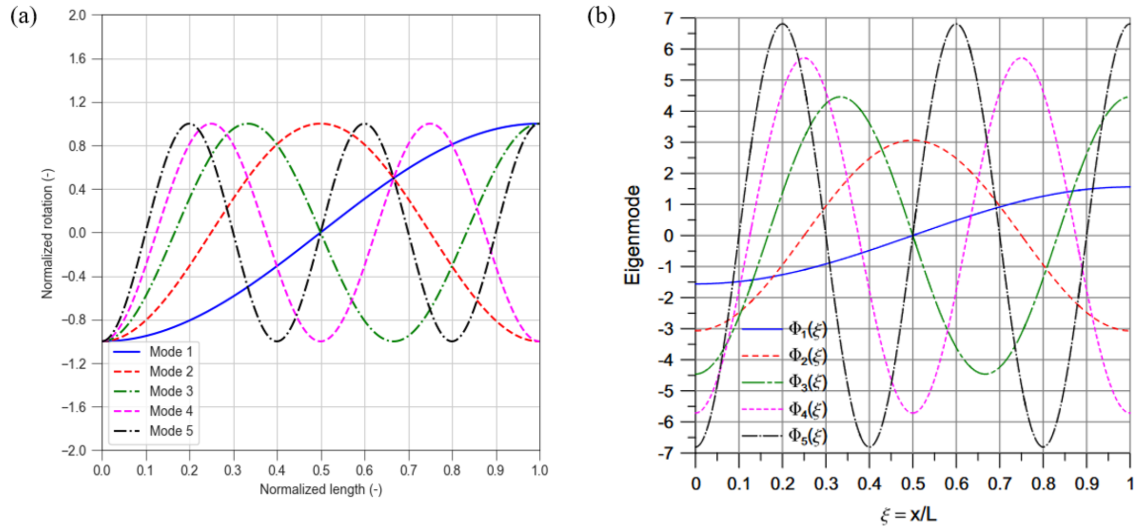

Figure A.4: A illustration of the first five mode shapes : (a.) shows the results of $\phi(x)$ obtained with the two-piece Timoshenko beam model and (b.) shows the results of the article [19]. The functions Φ_1, Φ_2, Φ_3, Φ_4 and Φ_5 are defined in article [19].

n	Eigenvalue code	Eigenvalue [19]
1	404.35408286	404.3540829
2	1597.56095701	1597.560957
3	3524.34808222	3524.348082
4	6104.92032031	6104.920320
5	9247.9937426	9247.993743
6	12861.93645177	1286.193645
7	16862.12382933	16862.12383
8	21174.58318127	21174.58318
9	25736.94980872	25736.94981
10	30497.85748609	30497.85749
11	35415.60970542	335415.60971
12	40456.65008636	40456.65009
13	45594.10053964	45594.10054
14	50806.48034954	50806.48035
15	56076.63514005	56076.63514
16	61390.86478017	61390.86478
17	66738.22380619	66738.22381
18	72109.96464699	72109.96465
19	77499.09603988	77499.09604
20	82900.0330413	82900.03304
21	88308.3193316	88308.31933
22	93720.40640408	93720.40640
23	99133.47749818	99133.47750
24	104545.30677815	104545.3068
25	109954.14634516	109954.1463

Table A.4: The eigenvalues (in rad/s) of the first part of the spectrum. The first column contains the results using a two-piece Timoshenko beam model. The second column contains the results of article [19].

Case 2: $\omega = \sqrt{\frac{kGA}{\rho I}}$

The transition frequency has with this example a value of $\omega_c = 111803.399$ (rad/s). The frequency is imposed, not solved. To obtain a non-trivial solution at the transition frequency, the determinant of the coefficient matrix at the transition frequency has to be equal to zero. In the case of the simply supported beam, the determinant of the coefficient matrix is always zero at the transition frequency:

n	Eigenvalue code	Eigenvalue [19]
26	111803.398874989	111803.3989

Table A.5: The eigenvalues (in rad/s) at the transition frequency. The first column contains the results using a two-piece Timoshenko beam model. The second column contains the results of article [19].

The mode obtained is a *pure shear vibration mode*, as shown in blue in Figure A.5 and A.6, where the transversal displacement is zero. Dependent on the rank of the coefficient matrix, a double eigenvalue can occur, indicated in red in Figures A.5 and A.6 as an example. This is not a pure shear vibration mode.

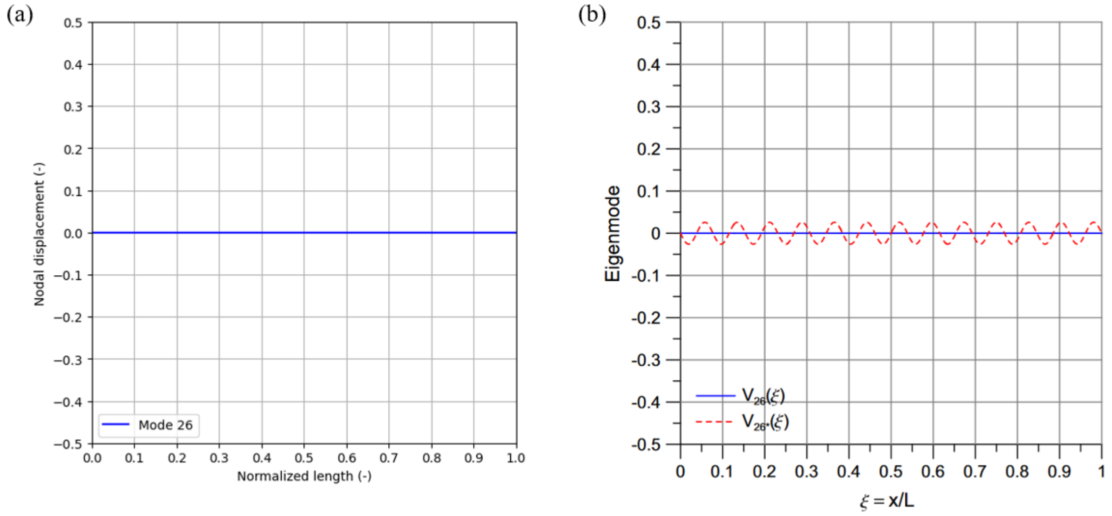

Figure A.5: The plotted mode shape at the transition frequency: (a.) shows the result of $Y(x)$ obtained with the two-piece Timoshenko beam model and (b.) shows the result of the article [19]. A double eigenvalue can occur at the transition frequency (an example of that is shown in red by the article) [19]. The functions V_{28} and $V_{28}*$ are defined in article [19].

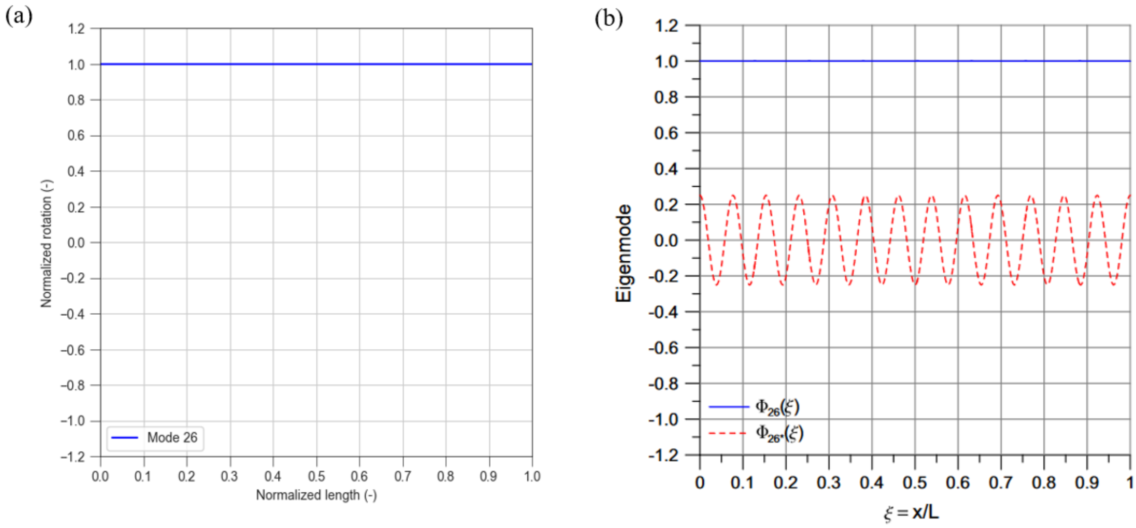

Figure A.6: The plotted mode shape at the transition frequency: (a.) shows the result of $\Psi(x)$ obtained with the two-piece Timoshenko beam model and (b.) shows the result of the article.A double eigenvalue can occur at the transition frequency (an example of that is shown in red by the article) [19]. The functions Φ_{28} and $\Phi_{28}*$ are defined in article [19].

Case 3: $\omega > \sqrt{\frac{kGA}{\rho I}}$

In the second part of the spectrum, the first 24 eigenvalues are obtained and given in table A.6. The obtained eigenvalues with the two-piece Timsohenko beam model show no differences with article [19], confirming the accuracy of the Python code for the second part of the spectrum.

n	Eigenvalue code	Eigenvalue [19]
27	112275.23826886	112275.2383
28	113670.65725684	113670.6573
29	115358.63529931	115358.6353
30	115933.65622448	115933.6562
31	118983.24267197	118983.2427
32	120757.72633166	120757.7263
33	122726.48603937	122726.4860
34	126150.62630367	126150.6263
35	127069.68674956	127069.6867
36	131536.74802825	131536.7480
37	131925.76578166	131925.7658
38	136915.67105325	136915.6711
39	137217.94847072	137217.9485
40	142287.10970443	142287.1097
41	142880.76318863	142880.7632
42	147650.88700103	147650.8870
43	148859.47650081	148859.4765
44	153006.91333603	153006.9133
45	155108.80982765	155108.8098
46	158355.1690327	158355.1690
47	161591.45479719	161591.4548
48	163695.6900625	163695.6901
49	168276.65482621	168276.6548
50	169028.556347	169028.5563

Table A.6: The first 24 eigenvalues (in rad/s) of the second spectrum. The first column contains the results using a two-piece Timoshenko beam model. The second column contains the results of article [19].

When the first six eigenmodes of the second spectrum are plotted in Figures A.7, A.8, A.9 and A.10, both low and high wavelengths appear in arbitrary order. The presence of lower wavelengths in the second part of the spectrum was initially observed by Traill-Nash and Collar [14], and was referred to as the second spectrum of modes. The shapes of these eigenmodes are identical to those appearing in the first part of the spectrum, however, they are associated with higher frequencies.

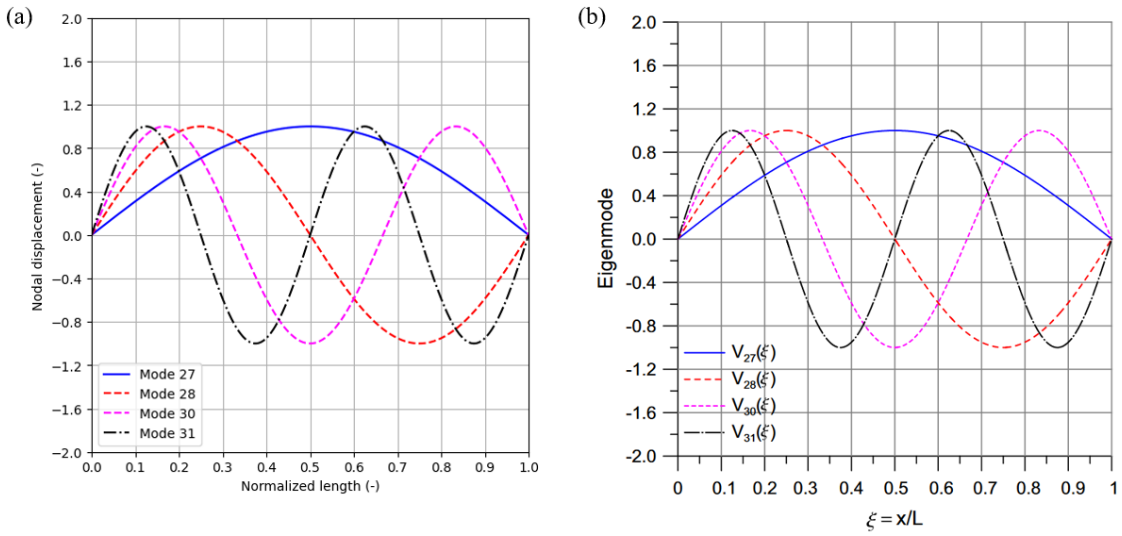

Figure A.7: The plotted mode shapes of modes 27, 28, 30 and 31: (a.) shows the results of $Y(x)$ obtained with the two-piece Timoshenko beam model and (b.) shows the results of the article [19]. The functions V_{27}, V_{28}, V_{30}, V_{31} are defined in article [19]. These mode shapes are part of the so-called second spectrum of modes, where the shape of the eigemode is identical to those appearing at the first part of the spectrum, however, they are associated with higher wave numbers.

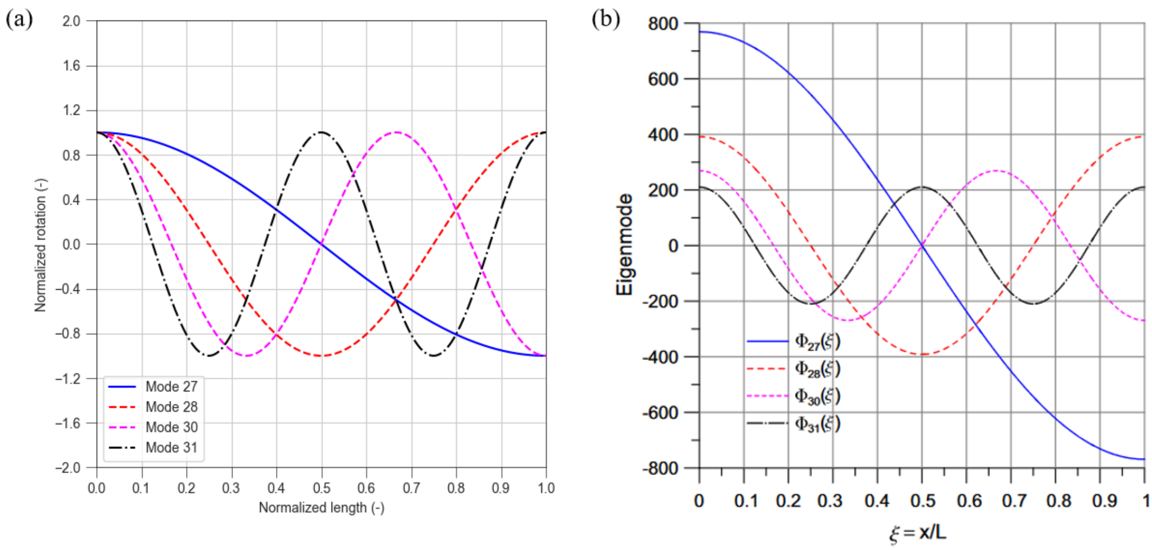

Figure A.8: The plotted mode shapes of modes 27, 28, 30 and 31: (a.) shows the results of $\Psi(x)$ obtained with the two-piece Timoshenko beam model and (b.) shows the results of the article [19]. The functions Φ_{27}, Φ_{28}, Φ_{30}, Φ_{31} are defined in article [19]. These mode shapes are part of the so-called second spectrum of modes, where the shape of the eigemode is identical to those appearing at the first part of the spectrum, however, they are associated with higher wave numbers.

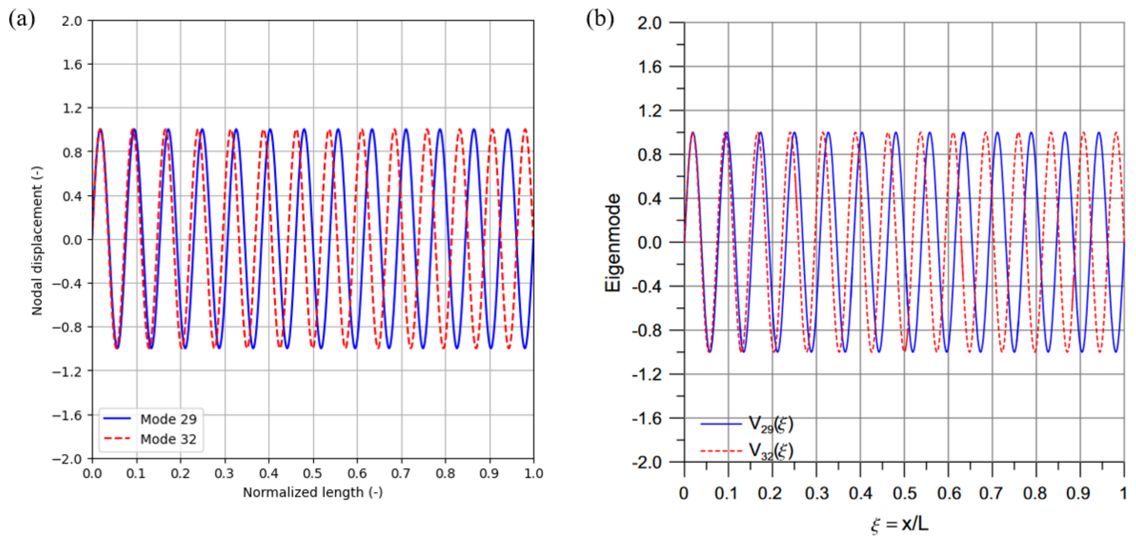

Figure A.9: The plotted mode shapes of modes 29 and 32: (a.) shows the results obtained with the two-piece Timoshenko beam model and (b.) shows the results of the article [19]. They are not part of the second spectrum of modes as these modes have no equivalent at lower frequencies. The functions V_{29}, and V_{32} are defined in article [19].

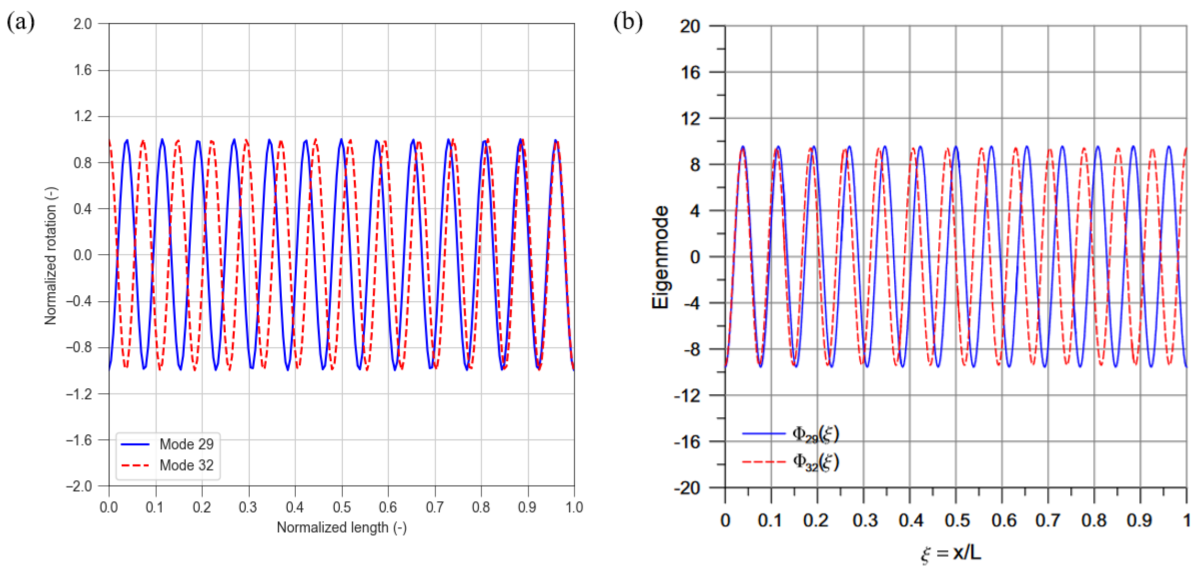

Figure A.10: The plotted mode shapes of modes 29 and 32: (a.) shows the results of $\Psi(x)$ obtained with the two-piece Timoshenko beam model and (b.) shows the results of the article [19]. They are not part of the second spectrum of modes as these modes have no equivalent at lower frequencies. The functions Φ_{29}, and Φ_{32} are defined in article [19].

A.7. Conclusion

With the Appendix, a thorough analysis of the discrete Timoshenko beam model is provided in the context of free vibrations, with a particular focus on the simply supported beam. This analysis brought insights into the nature of the vibration spectrum of a Timoshenko beam model. Despite numerous papers published since 1921, a complete solution has often been lacking, therefore failing to address the potential existence of a second spectrum of modes.

The analysis indicates the presence of a transition frequency, which separate the frequency spectrum into two parts.

The eigenmodes below and above this transition frequency exhibit difference in behavior.
To end, it should be noted that the second spectrum of modes is physically significant. The modes in this spectrum share the same wavelength, even if their frequencies differ. This means, for example, that a high-oscillating force can excite and potentially resonate with these modes. Ignoring these modes could lead to overlooking such phenomena. Only experimentation or possibly 3D modeling can validate the physical significance of these modes.

B

Verification Timoshenko beam model

In this research, the discrete Timoshenko beam model is chosen to approximate the dynamic behavior of the residential tower New Orleans and the office tower the Delftse Poort. To verify the code written (shown in Appendix F) and the behavior of the model, in this chapter, two verification studies are conducted. Section B.1 explains the two studies conducted. Section B.2 discusses the results obtained. The chapter ends with the conclusion of the results obtained.

B.1. Studies

Two studies are conducted. The first study verifies the code written for the discrete Timoshenko beam model and is explained in Section B.1.1. The second study verifies the model behavior and is explained in Section B.1.2.

B.1.1. Verification code model

To verify the discrete Timoshenko beam model, the uniform Timoshenko beam model of the study conducted by Taciroglu et al. [2] is replicated using the three-piece Timoshenko beam model. The beam model is shown in Figure B.1. The input parameter values of the uniform Timoshenko beam model are listed in Table B.1. Taciroglu et al. [2] consideration of the frequency dependency in the foundation is taken into account for this verification.

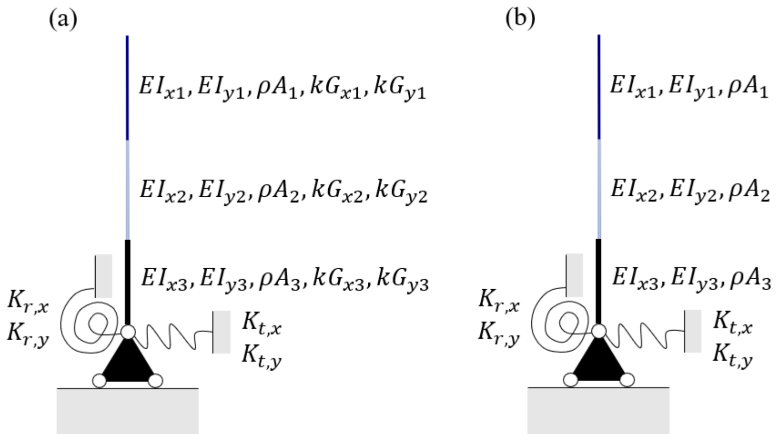

Figure B.1: The three-piece Timoshenko beam model (a) and the three-piece Euler-Bernoulli beam model (c).

B.1.2. Verification behavior model

To verify the behavior of the chosen model, the natural frequencies obtained with this three-piece Timoshenko beam model are compared with those obtained with a three-piece Euler-Bernoulli beam model, considering different values of the shear modulus G. The three-piece Euler-Bernoulli is shown

Properties segments	
L [m]	44
A [m^2]	843
I [m^4]	17750.25
ρ [kg/m^3]	400
E [N/m^2]	6.60e8
G [N/m^2]	2.75e8
k [-]	0.85
Properties boundary	
K_r [Nm/rad]	2.05e12
k_t [N/m]	1.84e10
Properties ground	
G_s [N/m]	2.68e8
ρ_s [kg/m3]	2.7e3

Table B.1: The input parameter values applied by Taciroglu et al. [2].

in Figure B.1. For this model, the same input parameter values (Table B.1) are applied. Taciroglu et al. [2] consideration of the frequency dependency is not taken into account for this verification.

B.2. Results

The results of the two studies conducted are shown below.

B.2.1. Verification code model

The mode shapes obtained with the three-piece Timoshenko beam model and by Taciroglu et al. [2] are shown in Figure B.2. The natural frequencies obtained with the three-piece Timoshenko beam model and by Taciroglu et al. [2] are listed in Table B.2. The modal properties obtained with the three-piece Timoshenko beam model show similarity to the properties obtained by Taciroglu et al. [2].

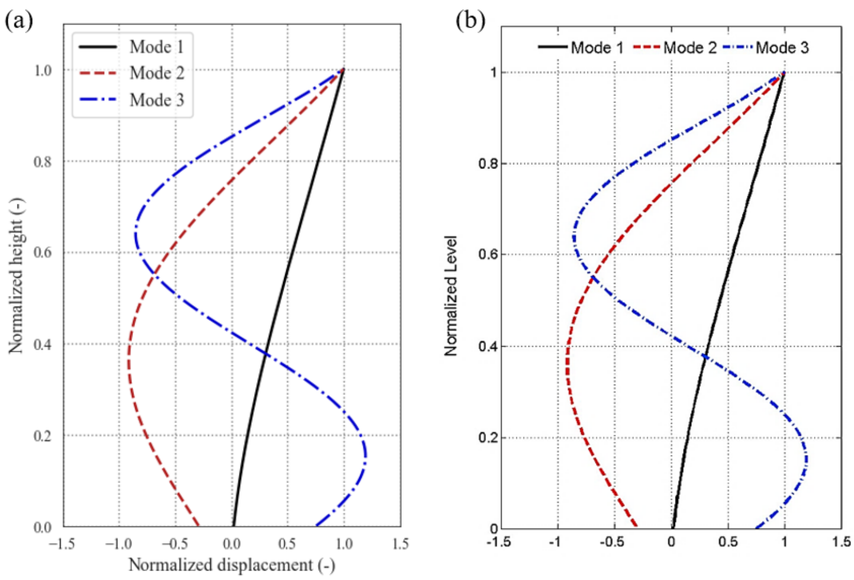

Figure B.2: An illustration of the first three mode shapes: (a) shows the results of transverse displacement obtained with the three-piece Timoshenko beam model, and (b) shows the results of the transverse displacement obtained by Taciroglu et al. [2].

Mode	Identified [Hz]	Values of article [2] [Hz]
1	1.608	1.61
2	6.687	6.68
3	14.234	14.22

Table B.2: The natural frequencies of the first three modes: (a) shows the results obtained with the three-piece Timoshenko beam model, and (b) shows the results from the article by Taciroglu et al. [2].

B.2.2. Verification behavior model

The natural frequencies obtained with the Timoshenko and Euler-Bernoulli beam models are shown in Figure B.3.

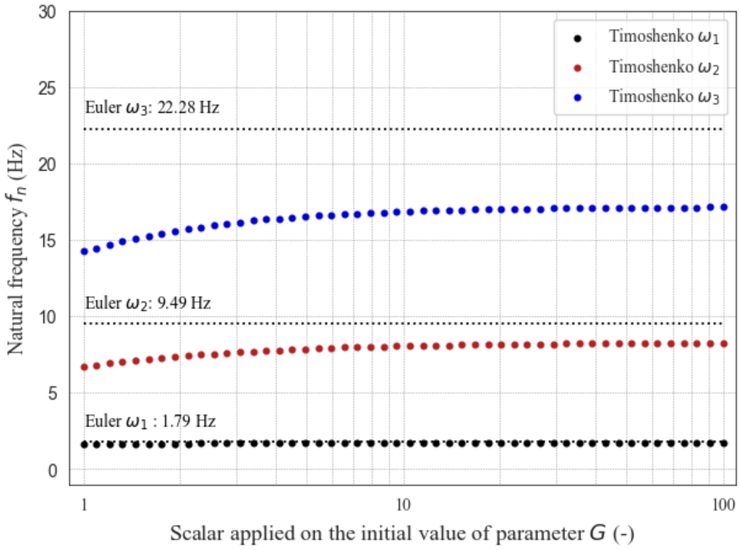

Figure B.3: The natural frequencies of the two models in question. The black dotted lines represent the natural frequencies of the Euler-Bernoulli beam model. The dots represent the natural frequencies of the Timoshenko beam model.

The first natural frequencies show similarities regardless of the value of G. However, differences can be observed at higher natural frequencies between the Timoshenko beam model and the Euler-Bernoulli beam model. As the value of the shear modulus G increases, the natural frequencies of the Timoshenko beam model tend to converge towards those of the Euler-Bernoulli beam model. Nonetheless, a constant difference remains, with the greatest difference occurring at the highest bending mode.

B.3. Conclusion

The results obtained by replicating the article by Taciroglu et al. [2] verify the code of the discrete Timoshenko beam model, as the results obtained with this model closely resemble the natural frequencies and mode shapes obtained by Taciroglu et al. [2].

The results obtained by comparing the Timoshenko beam model with the Euler-Bernoulli beam model verifies the behavior of the discrete Timoshenko beam model. The consistent difference in the natural frequencies are due to the consideration of rotational inertia. The Timoshenko beam model takes roational inertia in consideration, while the Euler-Bernoulli beam model does not take that into account. Rotational inertia measures the resistance to change in rotational motion, which is expected to be greater at higher modes, as there is more rotation at higher modes. Therefore, it is expected that the difference between the natural frequencies increases at higher modes. Moreover, as the shear modulus G increases, the natural frequencies of the Timoshenko beam model tend to converge towards those of the Euler-Bernoulli beam model. This is also expected, as with the Euler-Bernoulli beam model, no shear deformation can occur. This means that the Euler-Bernoulli beam model can be seen as having a shear modulus of ∞.

C

Structural Properties

Structural properties refer to intrinsic properties of the system, such as material properties (density, elasticity), geometric properties (dimensions, shapes), or boundary conditions [9], and form the basis of the discrete Timoshenko beam models used in this research. However, the values of the structural properties are not predefined. Therefore, in this appendix, the values are determined using the floor plans and Finite Element (FE) models of the residential tower New Orleans and the office tower Delfste Poort.

Section C.1 explains how each structural property is determined using the FE models and the floor plans of the New Orleans and the Delftse Poort. Sections C.2 and C.3 present the results obtained per building. The appendix ends with a conclusion.

C.1. Function description

With the floor plans of the buildings, the length L and the cross-sectional area A per segment are directly determined. The mass density of each segment ρ is calculated by considering the dead load G, variable load Q, and the volume of the segment:

$$\rho = \frac{G + 0.3 \times Q}{V} \tag{C.1}$$

Two approaches are considered to obtain the bending stiffness. The first approach calculates the bending stiffness per segment EI_s (with s being x or y) using the elastic modulus E_s applied in the FE model and the second moment of area I_s, calculated using the term $\frac{1}{12}bh^3$ and the Steiner rule:

$$EI_s = E_s \left(\frac{1}{12} bh^3 + bd^2 \right) \tag{C.2}$$

The FE segment approach considered the segments of the FE model. The FE model of the segment is clamped at the bottom, and a force F is applied at the top, resulting in a relative displacement Δ. The bending stiffness EI_s is then determined using this displacement Δ and Equation C.3:

$$EI_s = \frac{FL^3}{3\Delta} \tag{C.3}$$

The value of the shear modulus G_s is determined using the relationship between the shear modulus G_s and the elastic modulus E_s of the segment:

$$G_s = \frac{E_s}{2(1+\nu)} \tag{C.4}$$

The Poisson ratio ν is set to 0.2, which is typical for in-situ buildings. The shear coefficient k is taken as 0.85, in line with the study conducted by E. Taciroglu et al. [2]. The results are shown in Sections C.2 and C.3.

C.2. The residential tower New Orleans

The discrete Timoshenko beam model applied to approximate the dynamic behavior of the New Orleans is created using the FE model shown in Figure C.1. This FE model is divided into five segments (0 till IV). The segments were defined based on the floor plans. The floors within a segment have the same floor plan in the FE model. Therefore, a five-segment Timoshenko beam model is used to represent the FE model in Figure C.1. The floor plans of the segments are given in Figures C.2, C.3, C.4, C.5, and C.6.

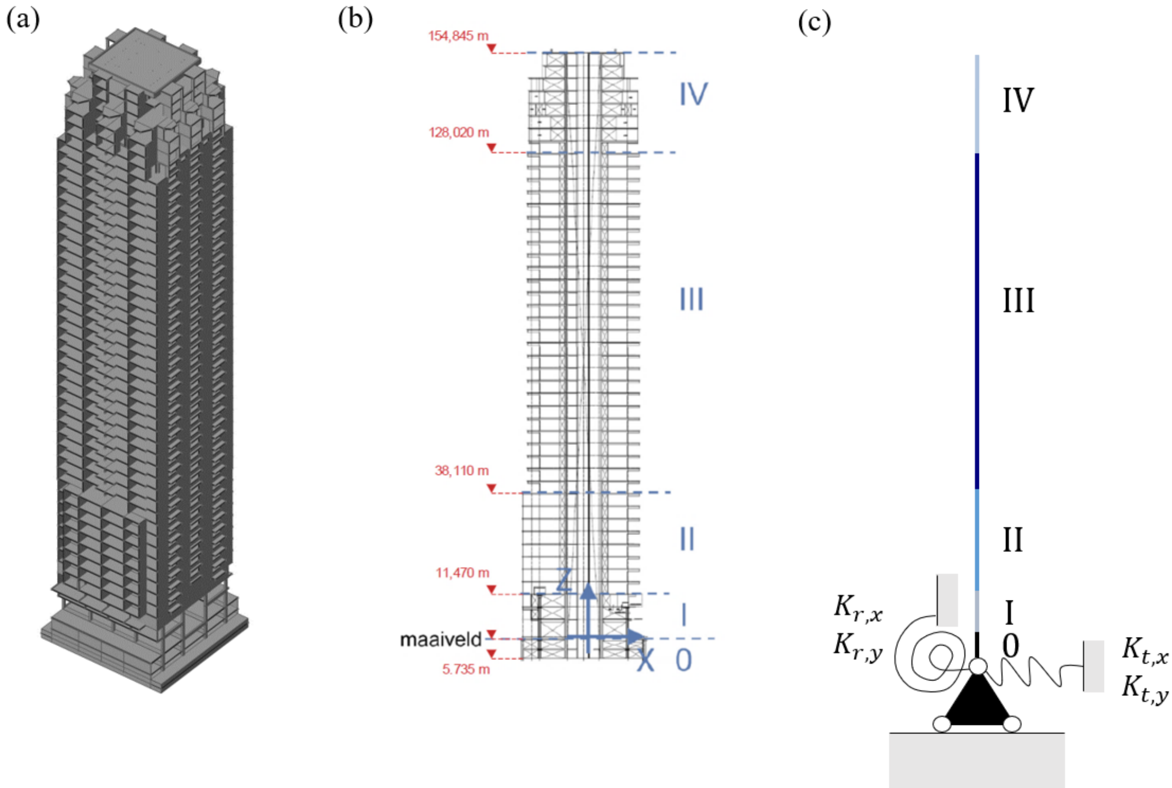

Figure C.1: Pictures of (a) the FE model of the New Orleans, (b) the 5 segments defined for the New Orleans model, and (c) the five-segment Timoshenko beam model.

The calculation of the bending stiffness using the Steiner approach is shown in Sections C.2.1, C.2.2, C.2.3, C.2.4, and C.2.5. The calculation of the other structural properties is shown in Section C.2.7.

C.2.1. Segment 0

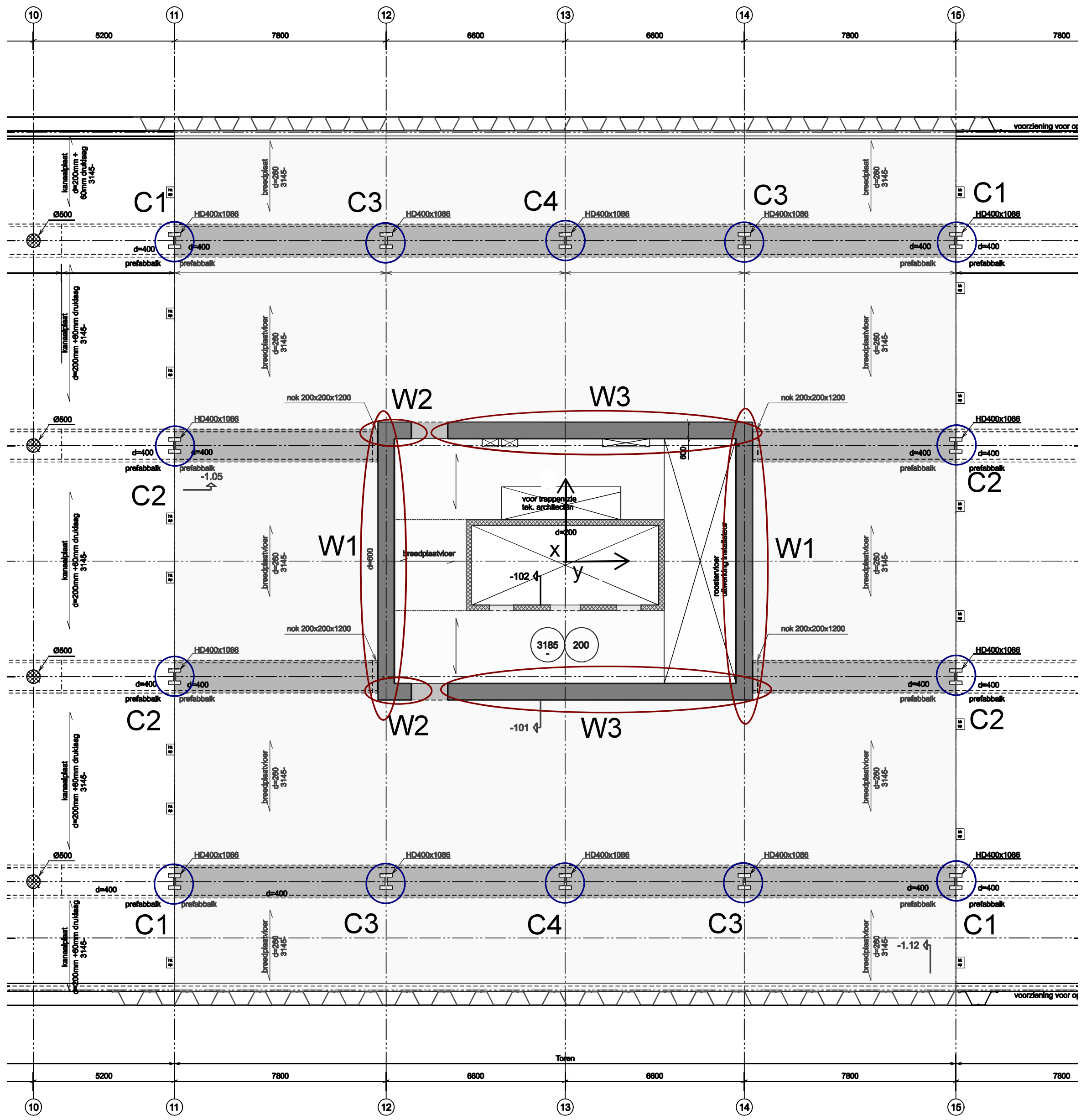

Figure C.2: The floor plan of segment 0.

For the Steiner approach, the second moment of area needs to be calculated. The second moment of area around the y-axis is denoted as I_y and is provided in Table C.1. The second moment of area around the x-axis is denoted as I_x and is provided in Table C.2. Both values are calculated using the floor plan shown in Figure C.2. The Steiner approach assumes rigid connection between the elements when calculating the second moment of area. Due to the lack of a rigid connection between the core and columns 1 and 2, indicated as C1 and C2 in Figure C.2, these columns are excluded from the calculation of I_x.

In the FEM model, $E_{columns}$ is equal to 210×10^9 N/m^2 and E_{walls} is equal to 38.2×10^9 N/m^2. This gives the bending stiffnesses $EI_y = 6.71 \times 10^{13}$ Nm^2 and $EI_x = 3.19 \times 10^{13}$ Nm^2.

Using the FE segment approach, the bending stiffnesses $EI_y = 5.87 \times 10^{13}$ Nm^2 and $EI_x = 2.73 \times 10^{13}$ Nm^2 are obtained.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_y$ [m^4]
Column 1 flange	8	0.454	0.125	11.65	0.000 591	61.618	61.619
Column 1 web	4	0.078	0.838	11.65	0.0153	35.477	35.492
Column 2 flange	4	0.454	0.125	4.45	0.000 296	4.495	4.495
Column 2 web	2	0.078	0.838	4.45	0.007 64	2.588	2.596
Column 3 flange	8	0.454	0.125	11.65	0.000 591	61.618	61.619
Column 3 web	4	0.078	0.838	11.65	0.0153	35.477	35.492
Column 4 flange	8	0.454	0.125	11.65	0.000 296	61.618	30.809
Column 4 web	4	0.078	0.838	11.65	0.007 64	35.477	17.746
Wall 1	2	0.6	8.4	0	59.270	0	59.270
Wall 2	2	1.295	0.6	4.75	0.0466	35.062	35.492
Wall 3	2	10.595	0.6	4.75	0.381	286.859	287.241
						Total	631.489

Table C.1: The second moment of area I_y of segment 0.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_x$ [m^4]
Column 3 flange	8	0.125	0.454	6.6	0.0078	19.776	19.784
Column 3 web	4	0.838	0.078	6.6	0.000 13	11.386	11.386
Wall 1	2	8.9	0.6	6.6	0.320	439.221	465.541
Wall 2	2	0.6	1.295	5.803	0.217	52.331	52.548
Wall 3	2	0.6	10.595	1.453	118.933	26.842	145.775
						Total	695.035

Table C.2: The second moment of area I_x of segment 0.

C.2.2. Segment I

The calculated second moment of area I_y is provided in Table C.3. The calculated second moment of area I_x is provided in Table C.4. Both are calculated using the floor plan shown in Figure C.3.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_y$ [m^4]
Column 1 flange	8	0.454	0.125	11.65	0.000 591	61.618	61.619
Column 1 web	4	0.078	0.838	11.65	0.0153	35.477	35.492
Column 2 flange	4	0.454	0.125	4.45	0.000 296	4.495	4.495
Column 2 web	2	0.078	0.838	4.45	0.007 64	2.588	2.596
Column 3 flange	8	0.454	0.125	11.65	0.000 591	61.618	61.619
Column 3 web	4	0.078	0.838	11.65	0.0153	35.477	35.492
Column 4 flange	8	0.454	0.125	11.65	0.000 296	61.618	30.809
Column 4 web	4	0.078	0.838	11.65	0.007 64	35.477	17.746
Wall 1	2	0.6	8.4	0	29.635	0	59.270
Wall 2	1	1.295	0.6	4.75	0.0233	17.531	17.554
Wall 3	1	10.595	0.6	4.75	0.191	143.4298	143.621
Wall 4	1	13.2	0.6	4.75	0.238	178.695	178.933
						Total	649.246

Table C.3: The second moment of area I_y of segment I.

In the FEM model, $E_{columns}$ is equal to 210×10^9 N/m^2 and E_{walls} is equal to 38.2×10^9 N/m^2. This gives the bending stiffnesses $EI_y = 6.77 \times 10^{13}$ Nm^2 and $EI_x = 3.31 \times 10^{13}$ Nm^2. Using the FE segment approach the bending stiffnesses $EI_y = 3.66 \times 10^{13}$ Nm^2 and $EI_x = 3.26 \times 10^{13}$ Nm^2 are obtained.

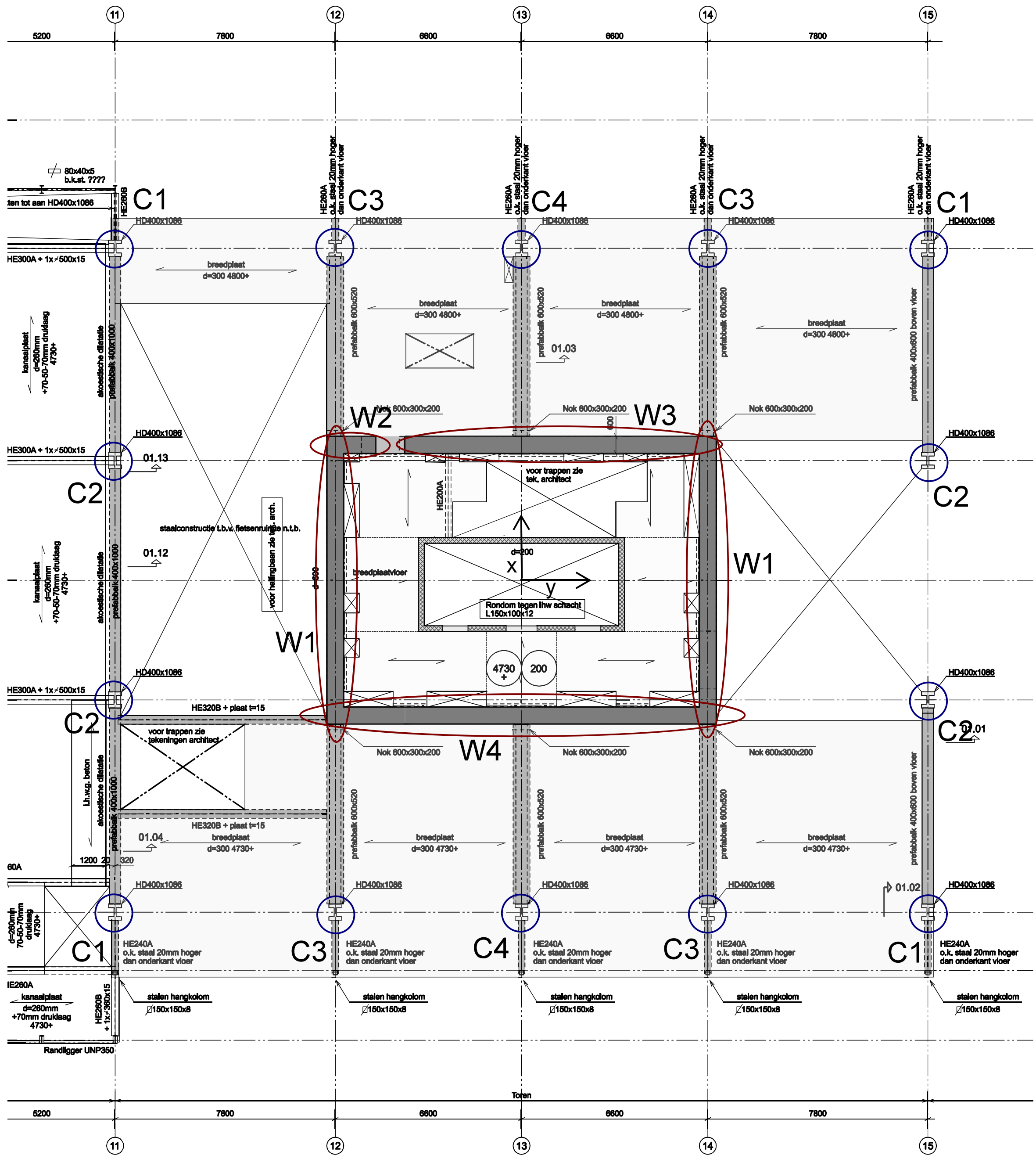

Figure C.3: The floor plan of segment I.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_x$ [m^4]
Column 3 flange	8	0.125	0.454	6.6	0.0078	19.776	19.784
Column 3 web	4	0.838	0.078	6.6	0.000 13	11.386	11.386
Wall 1	2	8.9	0.6	6.6	0.320	439.221	465.541
Wall 2	1	0.6	1.295	5.803	0.108	26.165	26.274
Wall 3	1	0.6	10.595	1.453	59.467	13.421	72.888
Wall 4	1	0.6	13.2	0	114.9984	0	131.719
						Total	727.592

Table C.4: The second moment of area I_x of segment I.

C.2.3. Segment II

The calculated second moment of area I_y is provided in Table C.5. The calculated second moment of area I_x is provided in Table C.6. Both are calculated using the floor plan shown in Figure C.4.

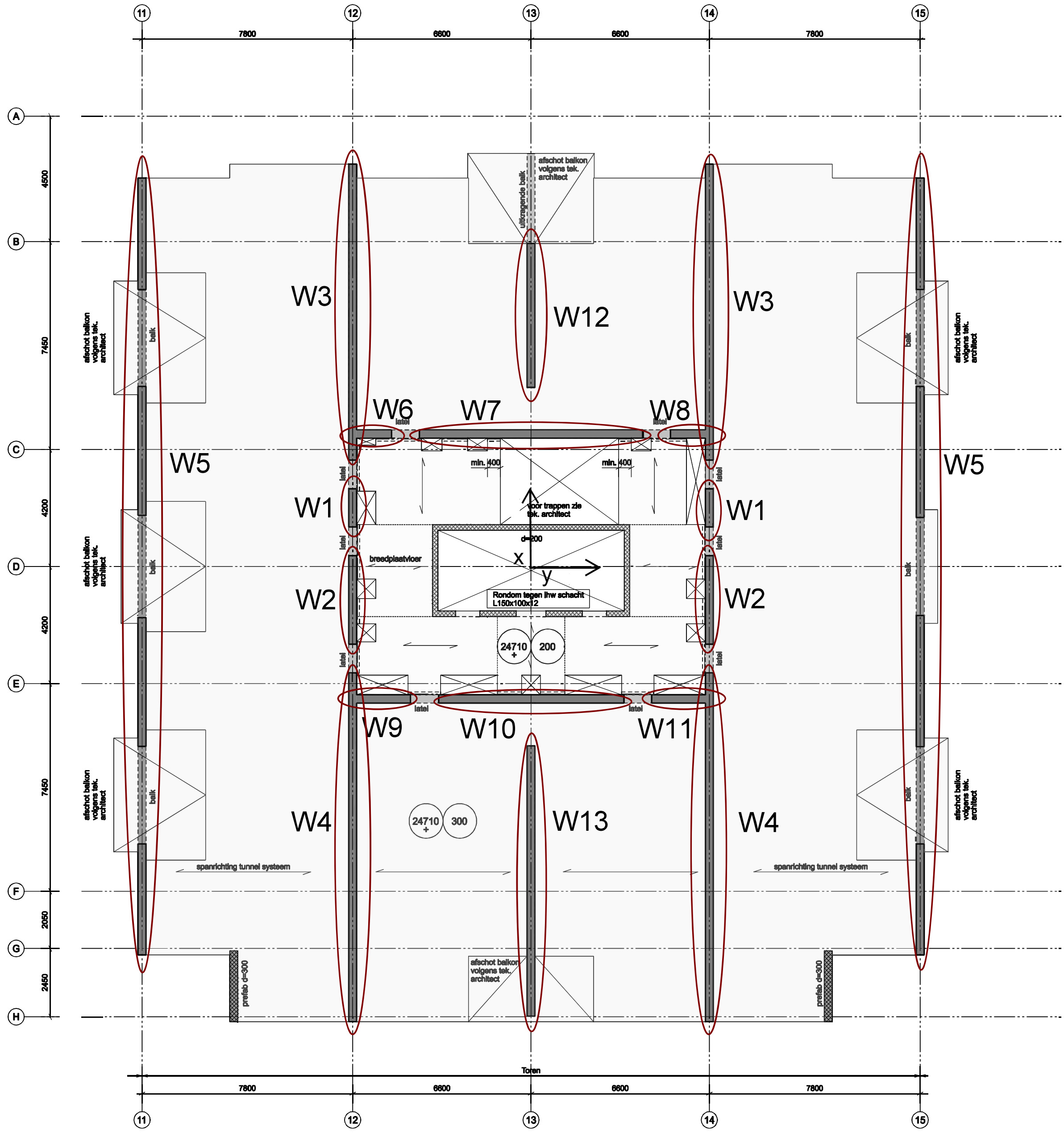

Figure C.4: The floor plan of segment II.

In the FEM model, $E_{columns}$ is equal to 210×10^9 N/m^2 and E_{walls} is equal to 38.2×10^9 N/m^2. This gives the bending stiffnesses $EI_y = 1.22 \times 10^{14}$ Nm^2 and $EI_x = 3.08 \times 10^{13}$ Nm^2. Using the FE segment approach, Besix obtained the bending stiffnesses $EI_y = 4.76 \times 10^{13}$ Nm^2 and $EI_x = 1.04 \times 10^{14}$ Nm^2.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_y$ [m^4]
Wall 1	2	0.3	1.36	2.041	0.126	3.402	3.528
Wall 2	2	0.3	3.19	1.124	1.629	2.421	4.050
Wall 3	2	0.3	10.59	9.08	59.366	523.815	583.181
Wall 4	2	0.3	12.54	10.121	98.620	770.704	869.324
Wall 5	2	0.3	27.86	0	1081.218	0	1081.218
Wall 6	1	1.295	0.3	4.75	0.003	8.766	8.768
Wall 7	1	7.968	0.3	4.75	0.018	54.055	54.073
Wall 8	1	1.295	0.3	4.75	0.003	8.766	8.768
Wall 9	1	1.995	0.3	4.75	0.004	13.504	13.508
Wall 10	1	6.625	0.3	4.75	0.015	44.843	44.858
Wall 11	1	1.995	0.3	4.75	0.004	13.504	13.508
Wall 12	1	0.3	5.16	9.003	3.425	125.350	128.775
Wall 13	1	0.3	9.66	11.253	22.501	366.784	389.285
						Total	3202.844

Table C.5: The second moment of area I_y of segment II.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_x$ [m^4]
Wall 1	2	1.36	0.3	6.6	0.0061	35.571	35.577
Wall 2	2	3.19	0.3	6.6	0.0144	83.478	83.493
Wall 3	2	10.59	0.3	6.6	0.0477	276.754	276.802
Wall 4	2	12.54	0.3	6.6	0.0564	327.772	327.828
Wall 6	1	0.3	1.295	5.803	0.0543	13.083	13.137
Wall 7	1	0.3	7.968	0	12.647	0	12.647
Wall 8	1	0.3	1.295	5.803	0.0543	13.083	13.137
Wall 9	1	0.3	1.995	5.453	0.1985	17.797	17.995
Wall 10	1	0.3	6.625	0	7.2694	0	7.2694
Wall 11	1	0.3	1.995	5.453	0.1985	17.797	17.995
Wall 12	1	5.16	0.3	0	0.0116	0	0.012
Wall 13	1	9.66	0.3	0	0.0217	0	0.022
						Total	805.913

Table C.6: The second moment of area I_x of segment II.

C.2.4. Segment III

The calculated second moment of area I_y is provided in Table C.7. The calculated second moment of area I_x is provided in Table C.8. Both are calculated using the floor plan shown in Figure C.5.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_y$ [m^4]
Wall 1	2	0.3	3.495	3.153	2.135	20.847	22.982
Wall 2	2	0.3	3.195	1.198	1.631	2.751	4.382
Wall 3	4	0.3	9.530	9.515	86.552	1035.361	1121.913
Wall 4	2	0.3	5.155	9.003	6.849	250.7	257.550
Wall 5	2	0.3	27.860	0	1081.218	0	1081.218
Wall 6	2	0.3	1.095	4.353	0.066	12.449	12.515
Wall 7	1	1.295	0.3	4.75	0.003	8.766	8.768
Wall 8	1	10.595	0.3	4.75	0.024	71.715	71.739
Wall 9	1	1.995	0.3	4.75	0.004	13.504	13.508
Wall 10	1	9.895	0.3	4.75	0.022	66.977	66.999
						Total	2661.574

Table C.7: The second moment of area I_y of segment III.

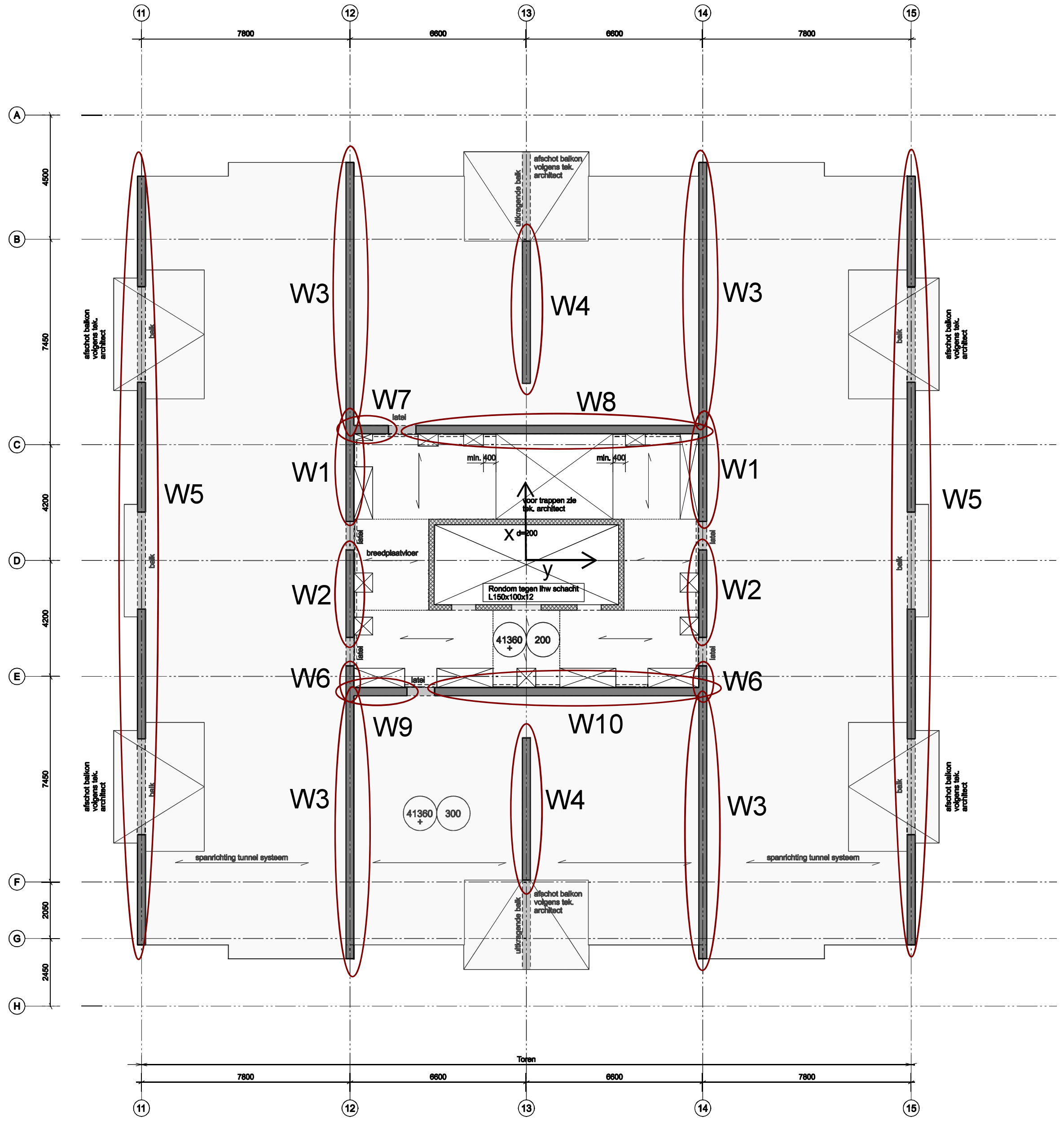

Figure C.5: The floor plan of segment III.

In the FEM model, $E_{columns}$ is equal to 210×10^9 N/m^2 and E_{walls} is equal to 38.2×10^9 N/m^2. This gives the bending stiffnesses $EI_y = 1.02 \times 10^{14}$ Nm^2 and $EI_x = 3.07 \times 10^{13}$ Nm^2.

Using the FE segment approach, Besix obtained the bending stiffnesses $EI_y = 4.59 \times 10^{13}$ Nm^2 and $EI_x = 8.92 \times 10^{13}$ Nm^2.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_x$ [m^4]
Wall 1	2	3.495	0.3	6.6	0.0157	91.345	91.361
Wall 2	2	3.195	0.3	6.6	0.0144	83.504	83.5189
Wall 3	4	9.530	0.3	9.515	0.0858	498.152	498.238
Wall 4	2	5.155	0.3	9.003	0.0232	0	0.0232
Wall 6	2	1.095	0.3	6.6	0.0049	28.619	28.624
Wall 7	1	0.3	1.295	5.803	0.0543	13.083	13.137
Wall 8	1	0.3	10.595	1.453	29.733	6.710	36.444
Wall 9	1	0.3	1.995	5.453	0.1985	17.797	17.995
Wall 10	1	0.3	9.895	1.803	24.221	9.650	33.870
						Total	803.211

Table C.8: The second moment of area I_x of segment III.

C.2.5. Segment IV

The calculated second moment of area I_y is provided in Table C.7. The calculated second moment of area I_x is provided in Table C.8. Both are calculated using the floor plan shown in Figure C.6.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_y$ [m^4]
Wall 1	1	0.3	3.490	5.59	1.063	32.717	33.779
Wall 2	1	0.3	5.561	0	4.299	0	4.2993
Wall 3	2	0.3	6.744	7.216	15.336	210.699	226.035
Wall 4	1	0.3	6.921	4.880	8.288	49.456	57.744
Wall 5	1	0.3	3.194	6.6	0.0072	41.739	41
Wall 6	2	13.2	0.3	4.75	0.0594	178.695	178.754
Wall 7	4	0.3	1.775	11.476	0.559	280.494	281.053
Wall 8	2	0.3	1.48	9.849	0.162	86.072	86.233
						Total	870.054

Table C.9: The second moment of area I_y of segment IV.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_x$ [m^4]
Wall 1	1	3.490	0.3	6.6	0.0079	45.607	45.615
Wall 2	1	5.561	0.3	6.6	0.0125	72.671	72.684
Wall 3	2	6.744	0.3	9.515	0.0303	176.261	176.292
Wall 4	1	6.921	0.3	6.6	0.0156	90.444	90.459
Wall 5	1	3.194	0.3	6.6	0.0072	41.739	41
Wall 6	2	0.3	13.2	0	114.998	0	114.998
Wall 7	4	1.775	0.2	6.6	0.0047	61.855	61.860
Wall 8	2	1.479	0.3	6.6	0.0067	38.655	38.662
						Total	803.211

Table C.10: The second moment of area I_x of segment IV.

In the FEM model, $E_{columns}$ is equal to 210×10^9 N/m^2 and E_{walls} is equal to 32.8×10^9 N/m^2. This gives the bending stiffnesses $EI_y = 3.32 \times 10^{13}$ Nm^2 and $EI_x = 2.45 \times 10^{13}$ Nm^2.

Using the FE segment approach, Besix obtained the bending stiffnesses $EI_y = 2.73 \times 10^{13}$ Nm^2 and $EI_x = 2.97 \times 10^{13}$ Nm^2.

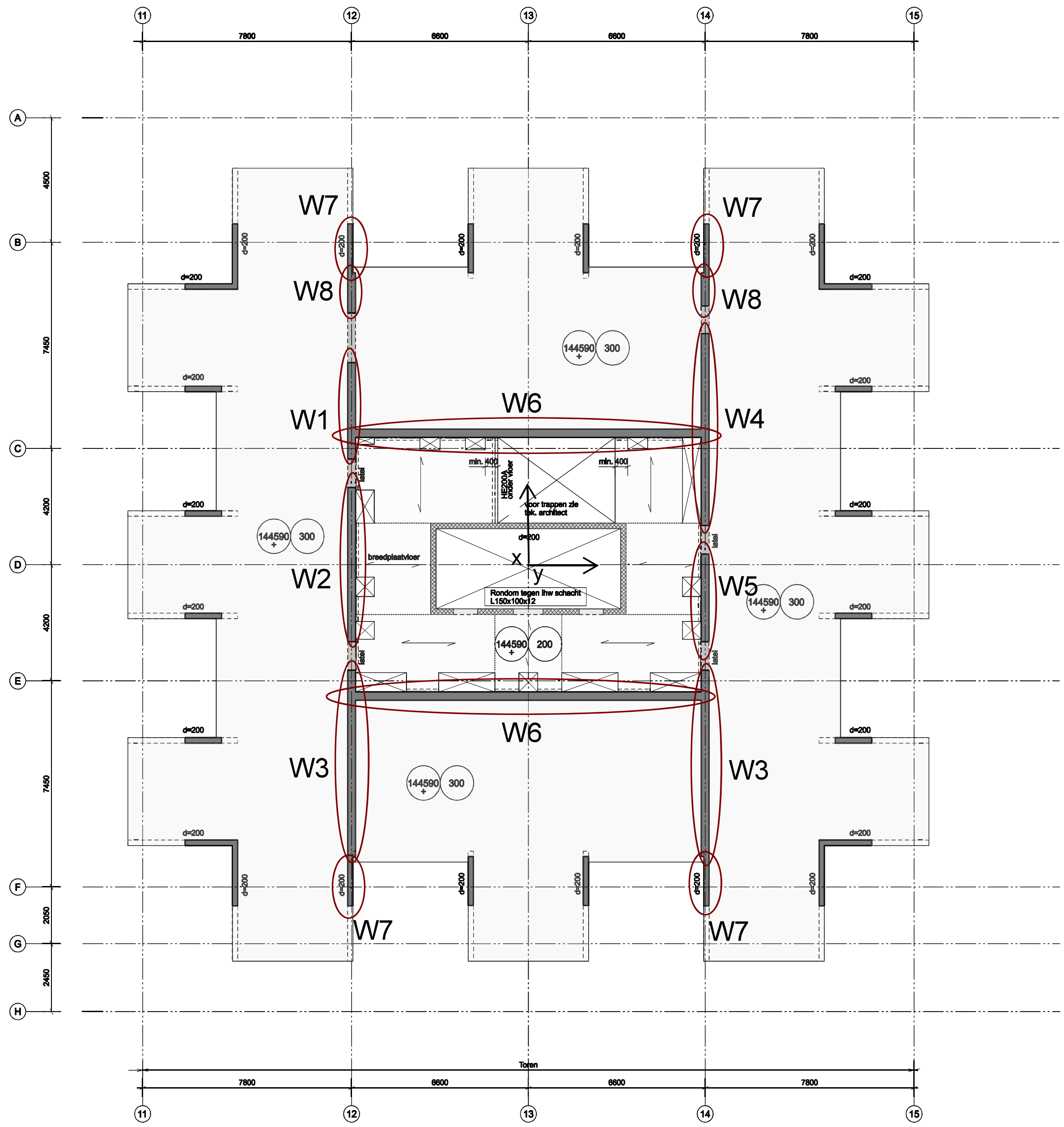

Figure C.6: The floor plan of segment IV.

C.2.6. Bending stiffness

The obtained bending stiffnesses are given in Table C.11. Additionally, the displacements at the top, using either a five-segment beam model with these bending stiffnesses or the FE model, are presented, when a force of 10,000 kN is applied and the base of the model is fixed.

The displacement in the y-direction obtained using the bending stiffnesses of Steiner approach EI_x shows significant differences. This could be due to not including columns 1 and 2 and wall 5 in I_x. Even when there is no rigid connection between these structural elements and the core, it can still be expected that these elements contribute to the rotational resistance. The actual contribution of these elements lies somewhere between the extremes of full inclusion and exclusion, but this contribution cannot be calculated using Steiner.

The displacement of the FE segment approach in the y-direction shows a closer resemblance to the displacement obtained with the Steiner approach. However, the stiffnesses are not as expected. Since the floor plans do not change significantly, it is unexpected to have such differences between the bending stiffnesses of the different segments. This approach fails to account for the intermediate relationships between the different segments (0 through IV) of the building. Segments 0 and I have column structures, which exhibit weaker behavior when considered individually than when integrated into the overall system. This is due to the additional mass from other segments. Therefore, considering these segment individually will lead to an inaccurate assessment of their bending stiffnesses.

Therefore, the bending stiffnesses of the Steiner approach are applied, with a correction factor included to account for the displacement differences between the FE model and five-segment beam using these segment stiffnesses. Since the FE model is stiffer in the y-direction, the segment stiffnesses EI_x are multiplied by $\frac{887}{516}$. Since the FE model is weaker in the x-direction, the segment stiffnesses EI_y are multiplied by $\frac{344}{394}$.

	EI$_x$ [Nm^2]		**EI$_y$ [Nm^2]**		**FEM**	
	Steiner	**FE segment**	**Steiner**	**FE segment**	**Y**	**X**
0	$3.19e13$	$2.37e13$	$6.71e13$	$5.87e13$		
I	$3.31e13$	$3.62e13$	$6.77e13$	$3.66e13$		
II	$3.08e13$	$1.04e14$	$1.22e14$	$4.76e13$		
III	$3.07e13$	$8.92e13$	$1.04e14$	$4.59e13$		
IV	$2.45e13$	$2.97e13$	$2.85e13$	$2.73e13$		
Δu [mm]	887	472	344	613	516	394

Table C.11: The bending stiffnesses and the total displacements determined with the different approaches.

C.2.7. Structural properties

The dead load and variable load provided by Besix are shown in Table C.12. The obtained structural properties are shown in Table C.13. The E modulus of the concrete is used as the E of the segments, as the building is almost entirely made of concrete. The second moment of area I of each segment is determined by dividing the bending stiffness by this E modulus.

Segment	**Dead load [kN]**	**Variable load $*0.3$ [kN]**
0	83621	1626
I	42088	1584
II	95562	3937
III	292545	11787
IV	76016	3058

Table C.12: The dead and variable load of the residential tower New Orleans.

Segment	0	I	II	III	IV
L [m]	5.735	11.47	26.640	89.91	26.825
A[m^2]	784	784	784	784	784
I_x [m^4]	1444	1500	1393	1389	1293
I_y [m^4]	1532	1548	2796	2324	760
ρ [kg/m^3]	1933	495	486	440	383
E [N/m^2]	38.2e9	38.2e9	38.2e9	38.2e9	32.8e9
v [-]	0.2	0.2	0.2	0.2	0.2
G [N/m^2]	15.9e9	15.9e9	15.9e9	15.9e9	13.7e9
k [-]	0.85	0.85	0.85	0.85	0.85
Boundary					
$K_{r,x}$ [GNm/rad]	2380	$K_{t,x}$ [GN/m]	19		
$K_{r,y}$ [GNm/rad]	2625	$K_{t,y}$ [GN/m]	4		

Table C.13: The structural property values obtained with the floor plans and FE model of the New Orleans.

C.3. The office tower the Delfste Poort

The discrete Timoshenko beam model chosen to approximate the dynamic behavior of the Delftse Poort is created using the Finite Element (FE) model of the building. This FE model, shown in Figure 6.1, is divided into four segments. The segements were defined based on the floorplans. Therefore, a four-segment Timoshenko beam model is used to represent the FE model in Figure 6.1. The floor plans representing the segments are shown in Figures C.8, C.9, and C.10. Since segments 0 and I have many similarities, the bending stiffness of both segments, determined using the Steiner approach, is calculated using the same floor plan. This is shown in Sections C.3.1, C.3.2, and C.3.3. The bending stiffness determined using the FE segment approach is also provided in those sections. The calculation of the other structural properties is shown in Section C.3.5.

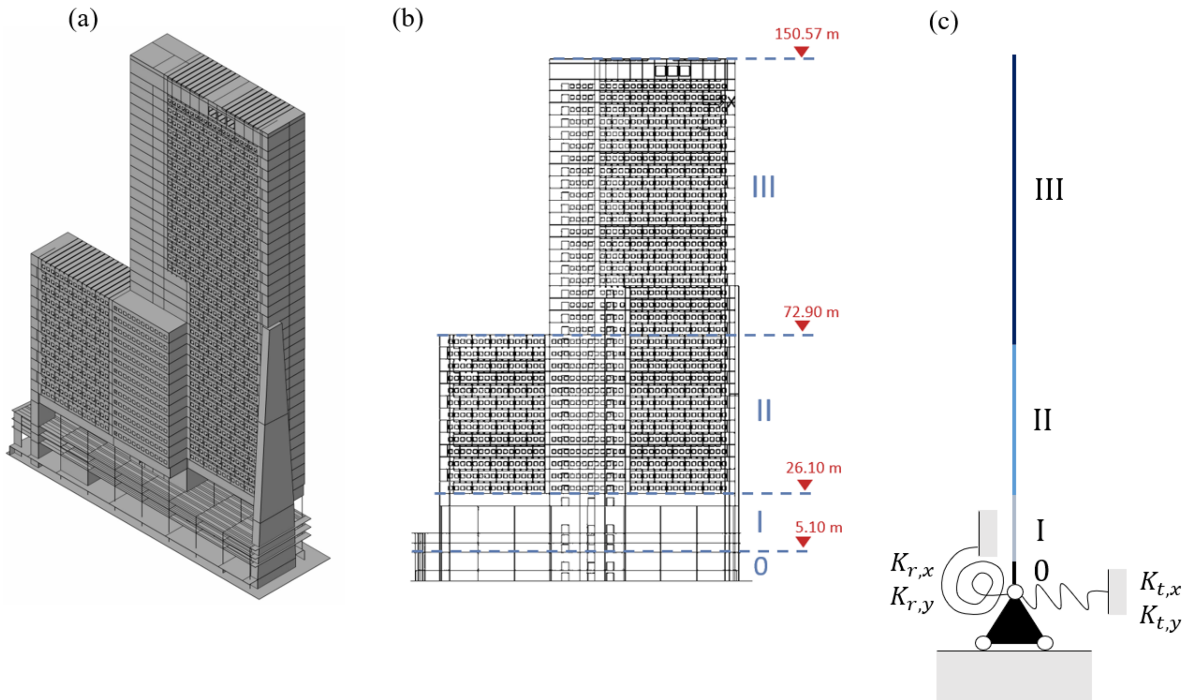

Figure C.7: Pictures of (a) the FE model of the Delftse Poort, (b) the 4 segments defined for the Delftse Poort model, and (c) the four-segment Timoshenko beam model.

C.3.1. Segment 0 and I

The calculated second moment of area I_x is provided in Table C.14. The calculated second moment of area I_y is provided in Table C.15. The percentage indicates the degree of closeness of the wall. The calculation takes this factor into account. Both are calculated using the floor plan shown in Figure C.8.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}bh^3$ [m^4]	Steiner [m^4]	I_x [m^4]
Wall 1	1	3.2	0.3	11.54	0.072	127.887	127.894
Wall 2	1	3.2	0.3	5.44	0.072	28.430	28.437
Wall 3	1	0.3	5.4	8.69	3.937	122.389	126.326
Wall 4	1	0.3	4.4	9.19	2.130	111.528	113.658
Wall 5	1	0.5	14.4	3.01	124.416	65.151	189.567
Wall 6	1	0.3	14.4	3.01	74.645	39.090	113.740
Wall 7	1	0.3	5.4	0.11	3.937	0.019	3.956
Wall 8	1	0.5	14.4	3.01	124.416	65.151	189.567
Wall 9	1	2.62	0.3	0.158	0.006	0.020	0.026
Wall 10	1	0.5	14.4	3.01	124.416	65.151	189.567
Wall 11 (90%)	1	23.0	0.3	4.34	0.047	117.071	117.112
Wall 12 (75%)	1	23.0	0.3	4.36	0.039	98.29	98.33
Wall 13	1	0.3	5.9	1.54	5.134	4.208	9.343
Wall 14	1	0.3	2.6	4.34	0.006	14.705	14.710
Wall 15	1	1.6	0.3	1.208	0.0036	0.701	0.704
Wall 16	1	0.4	4.85	6.97	3.803	94.163	97.966
Wall 17	1	0.4	2.6	2.76	0.586	7.911	8.497
Wall 18	1	0.4	8.5	0.19	20.470	0.125	20.596
Wall 19	1	0.4	13.35	2.67	79.309	37.980	117.289
Wall 20	1	3.1	0.4	2.67	0.017	23.014	23.031
Wall 21	1	3.1	0.4	9.64	0.017	115.278	115.295
Column 1	1	1.24	1.24	2.608	0.197	10.491	10.656
Column 2	1	1.24	1.24	10.99	0.197	185.776	185.973
Column 3	1	1.24	1.24	2.608	0.197	10.491	10.656
Column 4	1	1.24	1.24	10.99	0.197	185.776	185.973
Column 5	1	1.24	1.24	2.608	0.197	10.491	10.656
Column 6	1	1.24	1.24	10.99	0.197	185.776	185.973
Column 7	1	1.24	1.24	9.81	0.197	147.916	148.113
Column 8	1	1.24	1.24	10.99	0.197	185.776	185.973
Column 9	1	1.24	1.24	9.81	0.197	147.916	148.113
Column 10	1	1.24	1.24	10.99	0.197	185.776	185.973
Column 11	1	1.24	1.24	9.81	0.197	147.916	148.113
Column 12	1	1.24	1.24	3.79	0.197	22.108	22.305
Column 13	1	1.24	1.24	9.81	0.197	147.916	148.113
Column 12	1	1.24	1.24	3.79	0.197	22.108	22.305
Column 15	1	1.24	1.24	9.81	0.197	147.916	148.113
Column 16	1	1.24	1.24	9.81	0.197	147.916	148.113
						Total	3600.730

Table C.14: The second moment of area I_x of segment 0 and I.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}$bh^3 [m^4]	Steiner [m^4]	I$_y$ [m^4]
Wall 1	1	0.3	3.2	48.66	0.819	2272.864	2273.684
Wall 2	1	0.3	3.2	48.66	0.819	2272.864	2273.684
Wall 3	1	5.4	0.3	50.12	0.012	4067.458	4067.47
Wall 4	1	4.4	0.3	47.21	0.010	2941.702	2941.712
Wall 5	1	14.4	0.5	17.61	0.150	2232.211	2232.361
Wall 6	1	14.4	0.3	8.71	0.032	327.556	327.589
Wall 7	1	5.4	0.3	5.81	0.012	54.641	54.652
Wall 8	1	14.4	0.5	4.21	0.15	127.471	127.621
Wall 9	1	0.3	2.62	2.65	0.450	5.510	5.960
Wall 10	1	14.4	0.5	4.89	0.15	172.333	172.483
Wall 11 (90%)	1	0.3	23.0	6.357	273.758	251.007	524.764
Wall 12 (75%)	1	0.3	23.0	6.357	228.131	209.172	437.303
Wall 13	1	5.9	0.3	34.49	0.013	2105.808	2105.821
Wall 14	1	0.3	2.6	35.64	0.439	990.894	991.334
Wall 15	1	0.3	1.6	36.34	0.102	633.968	634.070
Wall 16	1	4.85	0.4	35.74	0.026	2478.380	2478.406
Wall 17	1	2.6	0.4	35.74	0.014	1328.616	1328.630
Wall 18	1	8.5	0.4	37.34	0.045	4741.134	4741.179
Wall 19	1	13.35	0.4	38.39	0.071	7871.013	7871.084
Wall 20	1	0.4	3.1	37.04	0.993	1701.448	1702.441
Wall 21	1	0.4	3.1	37.04	0.993	1701.448	1702.441
Column 1	1	1.24	1.24	49.56	0.197	3776.285	3776.482
Column 2	1	1.24	1.24	38.76	0.197	2309.714	2309.911
Column 3	1	1.24	1.24	38.76	0.197	2309.714	2309.911
Column 4	1	1.24	1.24	27.96	0.197	1201.835	1202.032
Column 5	1	1.24	1.24	27.96	0.197	1201.835	1202.032
Column 6	1	1.24	1.24	17.16	0.197	452.646	452.646
Column 7	1	1.24	1.24	17.16	0.197	452.646	452.646
Column 8	1	1.24	1.24	6.36	0.197	62.149	62.346
Column 9	1	1.24	1.24	6.36	0.197	62.149	62.346
Column 10	1	1.24	1.24	4.44	0.197	30.343	30.541
Column 11	1	1.24	1.24	4.44	0.197	30.343	30.541
Column 12	1	1.24	1.24	15.24	0.197	357.229	357.426
Column 13	1	1.24	1.24	15.24	0.197	357.229	357.426
Column 14	1	1.24	1.24	26.04	0.197	1042.806	1042.806
Column 15	1	1.24	1.24	26.04	0.197	1042.806	1042.806
Column 16	1	1.24	1.24	36.84	0.197	2087.075	2087.272
						Total	55 774.650

Table C.15: The second moment of area I_y of segment 0 and I.

In the FEM model, E_{walls} is equal to 11×10^9 N/m^2. This gives the bending stiffnesses $EI_x = 40 \times 10^{12}$ Nm^2 and $EI_y = 614 \times 10^{12}$ Nm^2. Using the FE segment approach, the bending stiffnesses $EI_x = 0.26 \times 10^{12}$ Nm^2 and $EI_y = 1.04 \times 10^{12}$ Nm^2 are obtained.

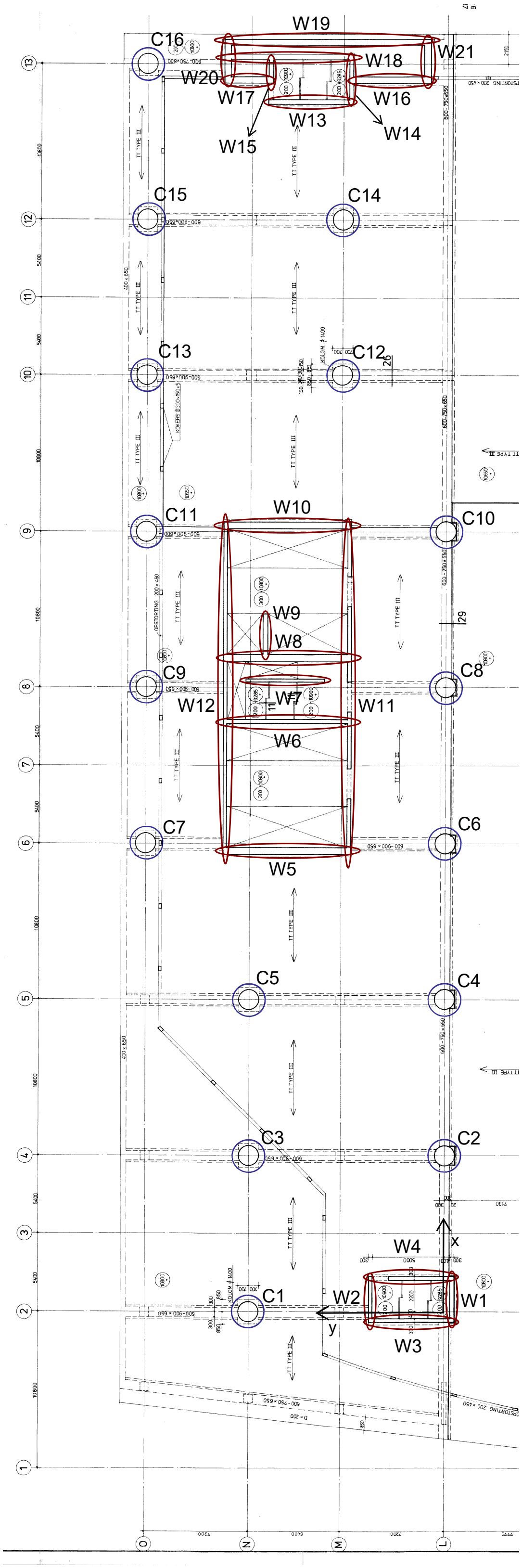

Figure C.8: The floorplan of segment 0 and I.

C.3.2. Segment II

The calculated second moment of area I_x is provided in Table C.16. The calculated second moment of area I_y is provided in Table C.17. The percentage indicates the degree of closeness of the wall. The calculation takes this factor into account. Both are calculated using the floor plan shown in Figure C.9.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}bh^3$ [m^4]	Steiner [m^4]	I_x [m^4]
Wall 1 (45%)	1	32.3	0.3	3.50	0.033	53.367	53.399
Wall 2 (45%)	1	55.3	0.3	11.20	0.056	936.746	936.801
Wall 3	1	0.3	14.4	3.85	74.650	64.088	138.737
Wall 4	1	0.3	5.4	8.35	3.937	112.995	116.931
Wall 5 (90%)	1	0.5	21.9	0.10	393.878	0.102	393.982
Wall 6 (60%)	1	22.0	0.3	10.70	0.030	453.242	453.272
Wall 7 (45%)	1	32.9	0.4	10.65	0.079	671.482	671.561
Wall 8 (75%)	1	22.0	0.3	4.00	0.037	79.265	79.302
Wall 9 (45%)	1	32.0	0.3	4.00	0.032	69.177	69.209
Wall 10 (75%)	1	22.0	0.3	4.70	0.037	109.270	109.307
Wall 11	1	2.62	0.3	1.85	0.006	2.685	2.691
Wall 12 (85%)	1	0.5	14.6	3.35	105.754	68.615	174.368
Wall 13 (90%)	1	0.5	21.6	0.25	377.914	0.615	378.529
Wall 14	1	0.3	5.5	1.10	4.159	2.002	6.162
Wall 15	1	0.4	14.3	3.30	97.474	62.229	159.703
Wall 16	1	0.4	4.3	6.45	2.650	71.593	74.243
Wall 17	1	0.4	13.0	2.10	73.233	22.968	96.201
Wall 18	1	0.7	0.4	4.15	0.003	4.819	4.822
Wall 19	1	2.3	0.4	8.40	0.012	64.940	64.953
						Total	3984.172

Table C.16: The second moment of area I_x of segment II.

Element	N	b [m]	h [m]	d [m]	$\frac{1}{12}bh^3$ [m^4]	Steiner [m^4]	I_y [m^4]
Wall 1 (45%)	1	0.3	32.3	33.39	379.106	4862.781	5241.886
Wall 2 (45%)	1	0.3	55.3	21.89	1902.514	3578.712	5481.226
Wall 3	1	14.4	0.3	49.39	0.032	10 539.980	10 540.020
Wall 4	1	5.4	0.3	46.49	0.012	3502.008	3502.020
Wall 5 (90%)	1	21.9	0.5	16.89	0.205	2812.836	2813.041
Wall 6 (60%)	1	0.2	22.0	5.64	159.72	126.165	285.885
Wall 7 (45%)	1	0.4	32.9	21.81	534.169	2815.806	3349.976
Wall 8 (75%)	1	0.3	22.0	5.64	199.650	157.706	357.356
Wall 9 (45%)	1	0.3	32.0	21.86	368.64	2063.514	2432.154
Wall 10 (75%)	1	0.3	22.0	5.64	199.650	157.706	357.356
Wall 11	1	0.3	2.62	1.93	0.450	2.941	3.391
Wall 12 (85%)	1	14.4	0.5	3.49	0.128	74.732	74.860
Wall 13 (90%)	1	21.6	0.5	5.61	0.203	305.425	305.627
Wall 14	1	5.5	0.3	35.11	0.012	2033.460	2033.473
Wall 15	1	14.3	0.4	38.056	0.076	8283.849	8283.926
Wall 16	1	4.3	0.4	36.406	0.023	2279.627	2279.650
Wall 17	1	13.0	0.4	39.16	0.069	7972.420	7972.489
Wall 18	1	0.4	0.7	38.61	0.011	417.309	417.320
Wall 19	1	0.4	2.3	37.76	0.406	1311.444	1311.849
						Total	57 043.499

Table C.17: The second moment of area I_y of segment II.

In the FEM model, E_{walls} is equal to 22×10^9 N/m^2. This gives the bending stiffnesses $EI_x = 88 \times 10^{12}$ Nm^2 and $EI_y = 1255 \times 10^{12}$ Nm^2. Using the FE segment approach, the bending stiffnesses $EI_x = 48.81 \times 10^{12}$ Nm^2 and $EI_y = 170.84 \times 10^{12}$ Nm^2 are obtained.

C.3.3. Segment III

The calculated second moment of area I_x is provided in Table C.18. The calculated second moment of area I_y is provided in Table C.19. The percentage indicates the degree of closeness of the wall. The calculation takes this factor into account. Both are calculated using the floor plan shown in Figure C.10.

Element	**N**	**b [m]**	**h [m]**	**d [m]**	$\frac{1}{12}$**bh^3 [m^4]**	**Steiner [m^4]**	**I$_x$ [m^4]**
Wall 1 (45%)	1	54.9	0.3	7.60	0.056	427.885	427.941
Wall 2 (50%)	1	54.9	0.3	6.30	0.062	327.034	327.096
Wall 3	1	0.5	15.0	0.25	140.625	0.462	141.087
Wall 4 (75%)	1	12.9	0.3	1.60	0.022	7.414	7.435
Wall 5	1	0.3	8.4	2.75	14.818	19.083	33.900
Wall 6 (85%)	1	0.5	14.4	0.25	105.754	0.377	106.131
Wall 7	1	0.3	5.5	4.202	4.159	29.131	33.290
Wall 8	1	0.4	14.4	0.45	99.533	0.355	99.888
						Total	1176.768

Table C.18: The second moment of area I_x of segment III.

Element	**N**	**b [m]**	**h [m]**	**d [m]**	$\frac{1}{12}$**bh^3 [m^4]**	**Steiner [m^4]**	**I$_y$ [m^4]**
Wall 1 (45%)	1	0.3	54.9	4.90	1861.528	177.954	2039.482
Wall 2 (50%)	1	0.3	54.9	4.90	2068.364	197.727	2266.091
Wall 3	1	15.0	0.5	22.80	0.156	3898.781	3898.937
Wall 4 (75%)	1	0.3	12.9	16.10	40.250	752.352	792.602
Wall 5	1	8.4	0.3	13.80	0.019	479.905	479.924
Wall 6 (85%)	1	14.4	0.5	9.40	0.128	540.757	540.884
Wall 7	1	5.5	0.3	29.20	0.012	1406.861	1406.874
Wall 8	1	14.4	0.4	32.15	0.077	5953.687	5953.764
						Total	17 378.560

Table C.19: The second moment of area I_y of segment III.

In the FEM model, E_{walls} is equal to 33×10^9 N/m^2. This gives the bending stiffnesses $EI_x = 39 \times 10^{12}$ Nm^2 and $EI_y = 573 \times 10^{12}$ Nm^2. Using the FE segment approach, the bending stiffnesses $EI_x = 41.60 \times 10^{12}$ Nm^2 and $EI_y = 255.56 \times 10^{12}$ Nm^2 are obtained.

C.3.4. Bending stiffness

The obtained bending stiffnesses are given in Table C.20. Additionally, the displacements at the top, using either a four-segment beam model with these bending stiffnesses or the FE model, are presented, when a force of 10,000 kN is applied and the base of the model is fixed.

With the FE segment approach, the bending stiffnesses vary significantly. The top segment, which represent the smallest part of the building, shows larger bending stiffness values then the bottom segments. These differences are not observed with the bending stiffnesses obtained using the Steiner approach. The displacements obtained with the Steiner approach show similarities with the displacements obtained using the FEM model.

The FE segment approach fails to account for the intermediate relationships between the different segments. For example, segments 0 and I have column structures, which exhibit weaker behavior when considered individually than when integrated into the overall system. This is due the additional mass from other segments. The E-moduli values of the different segments in the FE model may also contribute to the differences, as the top segment has a E-modulus value of 33×10^9 N/m^2), while the bottom segment has a E-modulus value of 11×10^9 N/m^2). However, it is not expected that this caused the factor of 100 difference between the segment stiffnesses.

Due to the differences in the segment stiffnesses obtained using the FE segment approach, the segment stiffnesses obtained using the Steiner approach are applied, with a correction factor included to account for the displacement differences between the FE model and four-segment beam using these segment stiffnesses. Since the FE model is weaker in the x-direction, the segment stiffnesses EI_y are multiplied by $\frac{16}{26}$. Since the FE model is stiffer in the y-direction, the segment stiffnesses EI_x are multiplied by $\frac{236}{199}$.

	EI$_y$ [Nm^2]		**EI$_x$ [Nm^2]**		**FEM**	
	Steiner	**FE segment**	**Steiner**	**FE segment**	**X**	**Y**
0	614e12	1.04e12	40e12	0.26e12		
I	614e12	6.46e12	40e12	5.53e12		
II	1255e12	170.84e12	88e12	48.81e12		
III	573e12	255.56e12	39e12	41.60e12		
Δu [mm]	16	-	236	-	26	199

Table C.20: The bending stiffnesses and displacements determined with the different approaches.

C.3.5. Structural properties

The dead load and variable load provided by Aronsohn are shown in Table C.21. The obtained structural properties are shown in Table C.22.

Segment	**Dead load [kN]**	**Variable load $*0.3$ [kN]**
0	202190	4641
I	154104	7811
II	241647	16538
III	228309	15722

Table C.21: The dead and variable load of the office tower the Delfste Poort.

Segment	0	I	II	III
L [m]	3.15	22.65	46.8	81.27
A[m^2]	1320	1320	1320	825
I_x [m^4]	4312	4312	4744	1402
I_y [m^4]	34350	34350	35105	10685
ρ [kg/m^3]	5071	552	426	371
E[N/m^2]	$11e9$	$11e9$	$22e9$	$33e9$
v [-]	0.2	0.2	0.2	0.2
G [N/m^2]	$4.6e9$	$4.6e9$	$9.2e9$	$1.38e9$
k [-]	0.85	0.85	0.85	0.85
Boundary				
$K_{r,x}$ [GNm/rad]	6486	$K_{t,x}$ [GN/m]	9.22	
$K_{r,y}$ [GNm/rad]	50371	$K_{t,y}$ [GN/m]	9.22	

Table C.22: The structural property values obtained with the floor plans and FE model of the Delftse Poort.

C.4. Conclusion

With this appendix, the analysis of obtaining the structural property values of the residential tower New Orleans and the office tower the Delftse Poort is provided. The analysis offers insights into how the translation from the FE (Finite Element) model to the discrete Timoshenko beam model is performed and how the different approaches affect the results obtained.

The analysis indicates difficulties obtaining values for the bending stiffness per segment. The Steiner approach, which calculated the bending stiffness per segment using the elastic modulus E and the second moment of area I (calculated with $\frac{1}{12}bh^3$ and the Steiner rule), assumed fully rigid connections between elements when determining I, making it impossible to accurately account for the contributions of elements that were not rigidly connected.

The FE approach, which used the relative displacement Δ of the FE segment to calculate the bending stiffness, did not account for the intermediate relationships between segments, leading to inaccurate estimation of segments with column structures. These segments exhibit weaker behavior when considered individually compared to when they were part of the overall system. This is due to the additional mass of the other segments.

Finally, it should be noted that the FE model also simplifies reality, and differences in behavior should be expected. The discrete Timoshenko beam model is more simplified then the FE model.

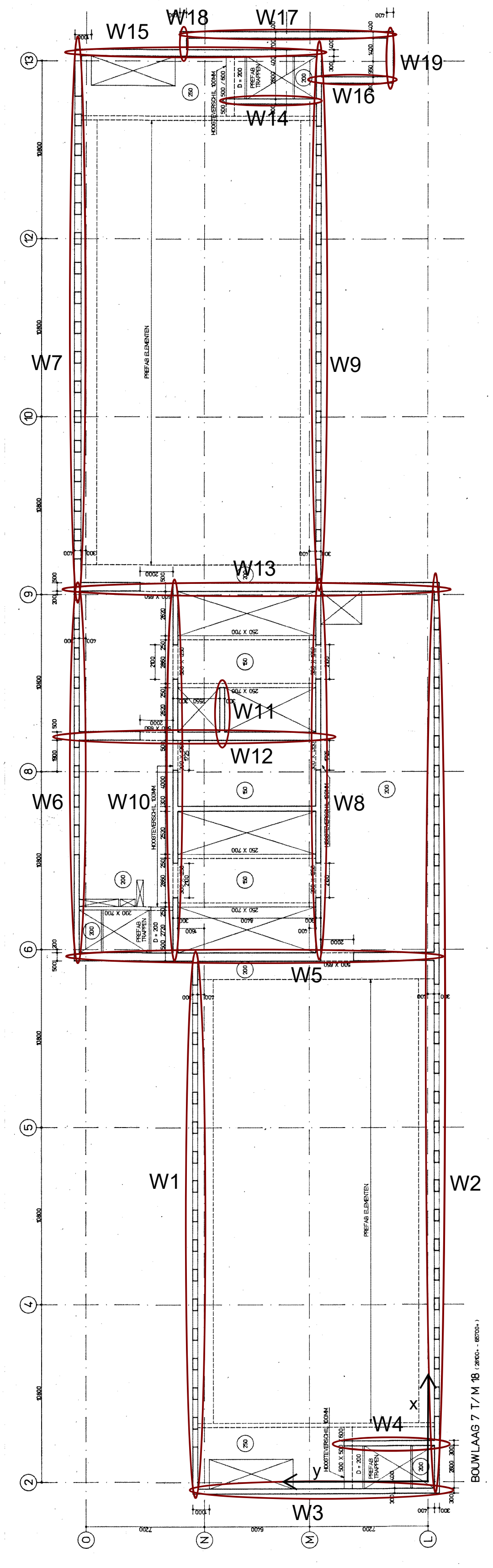

Figure C.9: The floor plan of segment II.

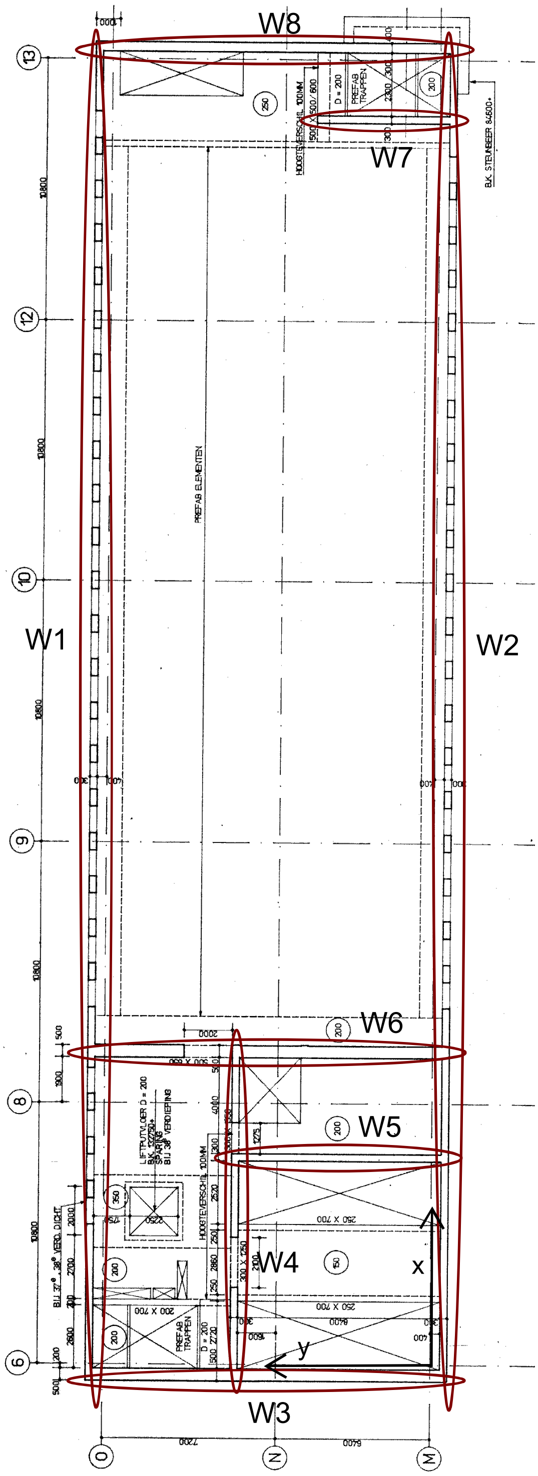

Figure C.10: The floor plan of segment III.

D

Measured modal properties

Modal properties refer to the characteristics of a physical system that describe its natural modes of vibration. These properties are important for understanding how a structure responds to dynamic loads and form the basis of the model update method applied in this research. However, these properties need to be determined first from vibration measurements.

Therefore, this appendix explains how the modal properties are determined from vibration measurements. It describes the measurement setup for the New Orleans and the Delftse Poort to obtain the vibration measurements and explains the data analysis used to determine the measured modal properties. The results are presented at the end.

D.1. Measurement setup the New Orleans

Since 2011, vibration measurements have been conducted on the residential tower New Orleans in Rotterdam, as shown in Figure D.1. The vibrations were monitored on the 34th floor using a permanent monitoring system. The system consisted of four acceleration sensors and was part of a research project focusing on local wind loads on facade elements and the influence of pressure equalization on these loads [17, 12].

Figure D.1: An picture of the residential tower New Orleans.

To determine the mode shapes of the building, for several months, additional acceleration sensors were installed on the 15th and 44th floors. The positions and orientations of the sensors are illustrated in Figure D.2 [17, 12].

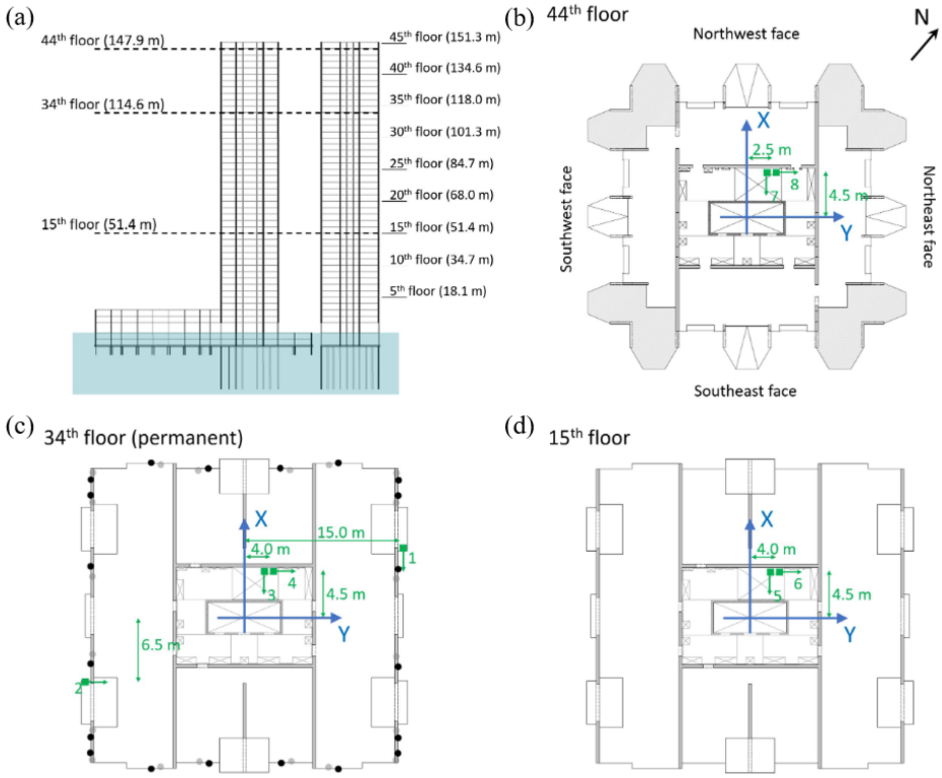

Figure D.2: Pictures of (a) the New Orleans and the positions of the acceleration sensors (green squares) on (b) the 44th floor, (c) the 34th floor, and (d) the 15th floor. The green arrows indicate the directions of the acceleration sensors.

D.2. Measurement setup the Delfste Poort

Vibration measurements are also conducted on the office tower the Delfste Poort, as shown in Figure D.3.

Figure D.3: An picture of the office tower the Delfste Poort.

The system consisted of two or three acceleration sensors on the 19th, 30th and 40th floor and floor 1. The positions and orientations of the sensors are illustrated in Figure D.4.

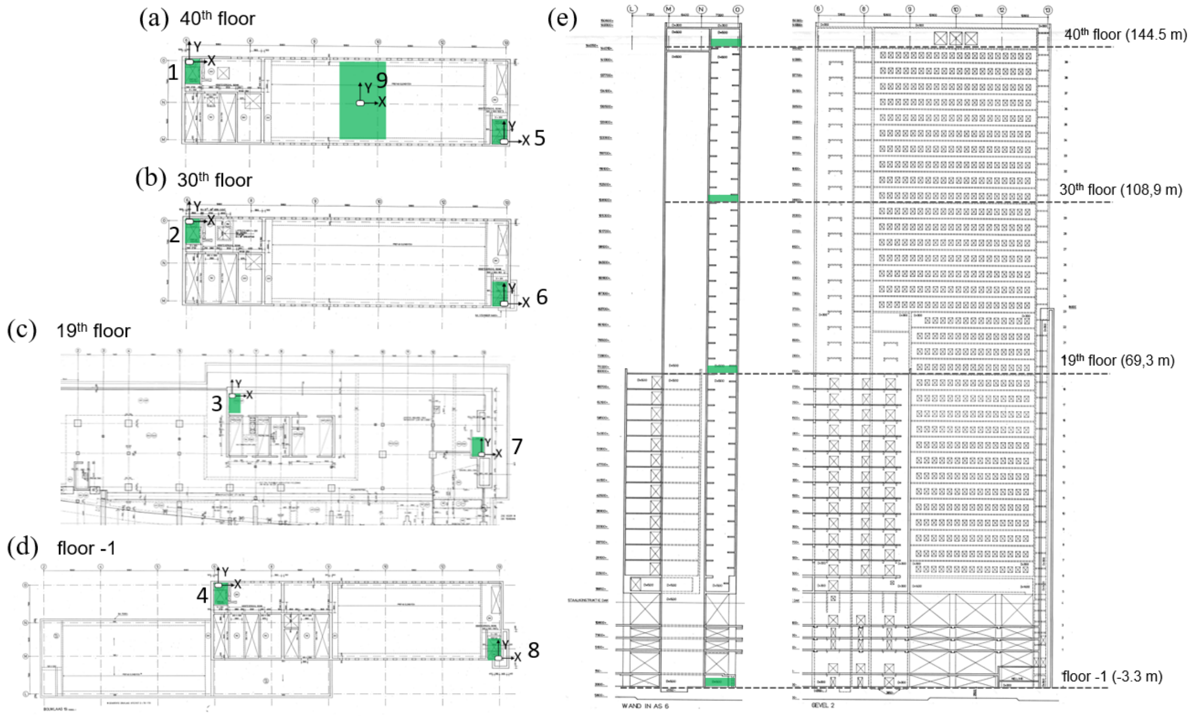

Figure D.4: Pictures of (e) the Delfste Poort and the positions of the acceleration sensors (green squares) on (d) floor 1, on (c) the 19th floor, (b) the 30th floor and (a) the 40th floor. The arrows indicate the directions of the acceleration sensors.

D.3. Data analysis

For the data analysis, the technique Frequency Domain Decomposition (FDD) is applied. The technique uses the relationship between the inputs $x(t)$ and the measured responses $y(t)$, expressed as [23]:

$$G_{yy}(j\omega) = \bar{H}(j\omega)G_{xx}(j\omega)H(j\omega)^T$$

where the $G_{xx}(j\omega)$ is the $(r\ x\ r)$ Power Spectral Density (PSD) matrix of the input, r is the number of inputs, $G_{yy}(j\omega)$ is the $(m\ x\ r)$ PSD of the responses, m is the number of responses and $H(j\omega)$ is the frequency response function. The overbar and superscript T respectively denote the complex conjugate and transpose of the frequency response function [23].

The inputs $x(t)$ are unknown. Therefore, to estimate the PSD matrix of the measured responses $y(t)$, white noise is assumed as input, which has a constant PSD matrix $G_{xx}(j\omega) = C$ [23]:

$$\hat{G}_{yy}(j\omega) = \bar{H}(j\omega)C(j\omega)H(j\omega)^T$$

The obtained estimate of the PSD $\hat{G}_{yy}$ is then decomposed at the each known frequency $\omega = \omega_i$ using Singular Value Decomposition:

$$\hat{G}_{yy}(j\omega_i) = U_i S_+ i U_i^H$$

Where the matrix U_i is the unitary matrix containing the singular vectors u_{ij} and S_i is the diagonal matrix containing the scalars singular values s_{ij} at each frequency instance ω [23]. The singular vectors u_{ij} can be associated with the mode shapes of the tested structure. The obtained mode shapes ϕ_i from the matrix U at the identified natural frequencies are scaled using Degree Of Freedom (DOF) scaling, defined as [12]:

$$\phi_{i,D} = \frac{\phi_i}{\phi_{i,Dn}}$$

where $\phi_{i,Dn}$ is the DOF with the largest component [24].

To see if the correlation between the obtained mode shapes is low (i.e if the obtained mode shapes are orthogonal), the Modal Assurance Criteria is applied, defined as:

$$MAC(\phi_i, \hat{\phi}_i) = \frac{|\phi_i \cdot \hat{\phi}_i|^2}{(\phi_i \hat{\phi}_i)(\phi_i \cdot \hat{\phi}_i)}$$

where ϕ_i and $\hat{\phi}_i$ are the compared mode shapes. If the MAC is high (larger than 35%), it can mean two things [24]:

- The modes are at close frequencies, indicating that the two modes are actually the same mode.
- The modes are at distance frequencies, indicating that the number of sensor positions is not sufficient to differentiate between the two modes.

The mode shapes that have a high MAC value are eliminated. The results of this analysis are given in Section D.4.

D.4. Results data analysis

The data applied of the New Orleans was recorded on 29/03/2020. A total of 129 10-minute records were used, which were sampled with 20 Hz [24]. The dimensionless spectrum of the first singular value is given in Figure D.5. The obtained modes are listed in Tables D.1 and D.2. The MAC values are given in Table D.3.

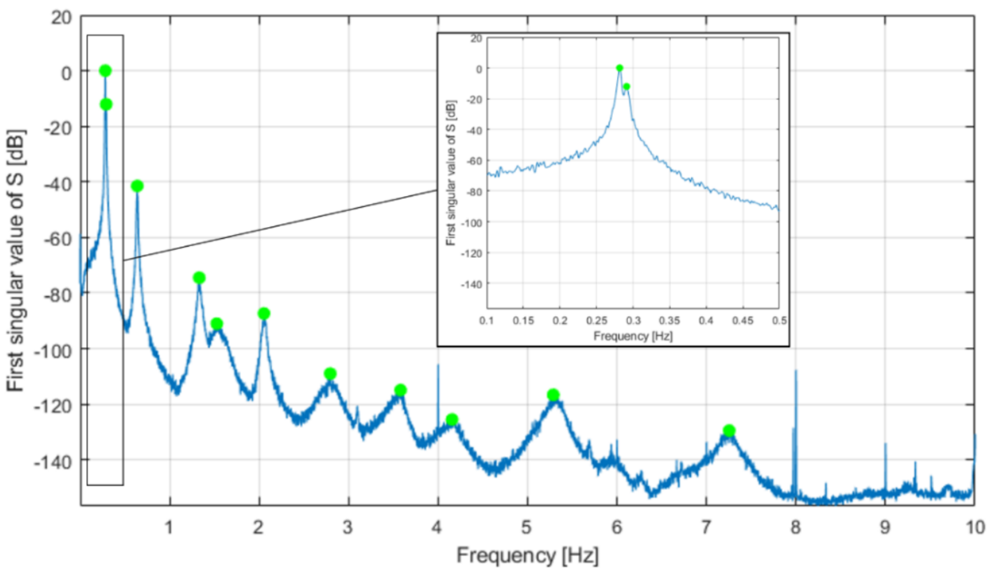

Figure D.5: The dimensionless spectrum of the first singular value. The green dots indicate the natural frequencies of the residential tower New Orleans [24].

According to Tables D.1 and D.2, for modes 1, 2, 3, 4, 5, 6, 7, and 9, a clear dominant direction is found. For the bending modes, relatively small modal displacements are found in the non-dominant direction. For modes 8, 10, and 11, however, no clear dominant direction can be determined from the modal displacements [24].

Table D.3 shows that modes 1, 2, 4, 5, 7, and 8 have MAC values lower than 35%. This indicates that the applied measurement setup is sufficient to observe these modes. Additionally, the table indicates the orthogonality of these modes [24].
Modes 3, 6, 9, 10, and 11 have more than 35% correlation with other modes. This indicates that the sensor setup is not sufficient to properly distinguish the mode shapes of these modes from the mode shapes of lower modes. However, it should be noted that when mode 3 is not considered, mode 6 does meet the 35% correlation criterion [24].

Mode	1	2	3	4	5	6
Dominant direction	X	Y	Θ	Y	X	Θ
Natural frequency [Hz]	0.282	0.291	0.638	1.332	1.527	2.054
Modal displacement						
Sensor 5 (51.4 m)	−0.30	−0.05	−0.13	−0.00	0.89	0.29
Sensor 6 (51.4 m)	−0.02	−0.29	−0.15	−0.90	0.05	0.50
Sensor 1 (114.6 m)	−0.71	−0.10	−1.00	−0.02	−0.23	−1.00
Sensor 2 (114.6 m)	−0.03	−0.78	−0.43	−0.19	−0.02	−0.45
Sensor 3 (114.6 m)	−0.67	−0.11	−0.28	−0.00	−0.06	−0.24
Sensor 4 (114.6 m)	−0.03	−0.76	−0.30	−0.06	−0.02	−0.21
Sensor 7 (147.9 m)	−1.00	−0.12	−0.23	−0.04	−1.00	−0.25
Sensor 8 (147.9 m)	−0.09	−1.00	−0.38	1.00	−0.11	−0.61

Table D.1: The identified measured modes (1-6) of the residential tower New Orleans (normalized).

Mode	7	8	9	10	11
Dominant direction	Y		Y		
Natural frequency [Hz]	2.771	3.560	4.155	5.300	7.250
Modal displacement					
Sensor 5 (51.4 m)	0.02	0.33	0.03	0.08	−0.16
Sensor 6 (51.4 m)	−0.73	0.22	0.18	−0.03	−0.06
Sensor 1 (114.6 m)	−0.11	−1.00	−0.12	−1.00	−1.00
Sensor 2 (114.6 m)	0.83	0.11	1.00	0.15	0.20
Sensor 3 (114.6 m)	−0.06	−0.80	−0.08	−0.58	−0.51
Sensor 4 (114.6 m)	0.99	−0.38	0.95	−0.29	−0.13
Sensor 7 (147.9 m)	0.01	0.41	−0.06	0.16	0.16
Sensor 8 (147.9 m)	−1.00	0.39	0.35	0.14	0.11

Table D.2: The identified measured modes (7-11) of the residential tower New Orleans (normalized).

Mode	1	2	3	4	5	6	7	8	9	10	11
1	1.00										
2	0.04	1.00									
3	0.44	0.06	1.00								
4	0.00	0.20	0.01	1.00							
5	0.24	0.01	0.06	0.00	1.00						
6	0.27	0.03	0.69	0.24	0.19	1.00					
7	0.00	0.00	0.09	0.00	0.00	0.04	1.00				
8	0.11	0.00	0.32	0.01	0.00	0.30	0.06	1.00			
9	0.00	0.73	0.01	0.05	0.00	0.02	0.26	0.00	1.00		
10	0.27	0.00	0.63	0.01	0.01	0.43	0.00	0.88	0.00	1.00	
11	0.30	0.00	0.68	0.02	0.00	0.39	0.00	0.74	0.02	0.94	1.00

Table D.3: The MAC of the bending modes of Tables D.1 and D.2. The MAC values higher then 0.35 are indicated in lightsteelblue.

The obtained modes for the office tower the Delfste Poort are listed in Tables D.4 and D.5. The MAC values are given in Table D.6. According to Tables D.4 and D.5, for modes 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10, a clear dominant direction is found.
For the bending modes, relatively small modal displacements are found in the non-dominant direction. For mode 11, however, no clear dominant can be determined from the modal displacements.

Table D.6 shows that modes 1, 2, 3, and 4 have MAC values below 35%. The applied measurements setup is sufficient to observe these modes. The orthogonality of these modes is also proven with this table. Modes 5, 6, 7, 8, 9, 10 and 11 have more than 35% correlations with other modes, indicating that the sensor setup is not sufficient to properly distinguish the mode shapes of these modes from the modes shapes of lower modes.

Mode	**1**	**2**	**3**	**4**	**5**	**6**
Dominant direction	Y	X	Θ	Y	X	Θ
Natural frequency [Hz]	0.403	0.861	1.165	1.624	2.027	2.454
Modal displacement						
Sensor 4 (−3.30 m)	0.00	−0.06	−0.05	0.02	0.28	−0.01
Sensor 4 (−3.30 m)	0.01	0.00	−0.03	0.01	0.02	−0.02
Sensor 3 (69.30 m)	−0.02	−0.48	0.00	0.03	0.48	0.01
Sensor 3 (69.30 m)	0.33	0.01	−0.51	0.37	−0.12	0.05
Sensor 7 (69.30 m)	−0.02	−0.49	0.13	0.04	0.44	0.06
Sensor 7 (69.30 m)	0.34	0.00	0.36	0.61	−0.19	0.39
Sensor 2 (108.90 m)	−0.02	−0.77	0.03	−0.05	−0.26	0.01
Sensor 2 (108.90 m)	0.65	0.00	−0.65	−0.19	−0.19	0.37
Sensor 6 (108.90 m)	−0.02	−0.75	0.23	0.03	−0.18	−0.09
Sensor 6 (108.90 m)	0.66	−0.03	0.71	0.42	0.12	−0.16
Sensor 1 (144.51 m)	−0.02	−1.00	0.06	−0.11	−1.00	−0.04
Sensor 1 (144.51 m)	0.96	−0.01	−0.68	−1.00	−0.01	0.55
Sensor 5 (144.51 m)	−0.06	−1.00	0.12	−0.05	−0.99	−0.09
Sensor 5 (144.51 m)	1.00	−0.10	1.00	−0.21	0.41	−1.00
Sensor 9 (144.51 m)	−0.03	−1.00	0.11	−0.08	−0.97	−0.09
Sensor 9 (144.51 m)	0.99	−0.06	0.58	−0.46	0.33	−0.59

Table D.4: The identified measured modes (1-5) of the office tower the Delfste Poort (normalized).

Mode	**7**	**8**	**9**	**10**	**11**
Dominant direction	X	Y	Θ	X	
Natural frequency [Hz]	2.962	3.749	4.493	4.939	5.844
Modal displacement					
Sensor 4 (−3.30 m)	0.02	−0.03	0.01	0.07	−0.14
Sensor 4 (−3.30 m)	0.01	−0.01	0.03	0.00	−0.21
Sensor 3 (69.30 m)	0.29	0.07	−0.03	0.63	0.13
Sensor 3 (69.30 m)	−0.05	0.18	0.58	−0.15	0.14
Sensor 7 (69.30 m)	0.29	0.06	−0.25	0.71	0.27
Sensor 7 (69.30 m)	0.06	−0.36	−0.70	0.44	0.35
Sensor 2 (108.90 m)	−0.52	0.03	0.05	0.74	0.78
Sensor 2 (108.90 m)	0.04	0.67	0.21	−0.03	−0.02
Sensor 6 (108.90 m)	−0.48	−0.10	−0.05	0.92	1.00
Sensor 6 (108.90 m)	−0.17	0.92	−0.62	0.08	−0.07
Sensor 1 (144.51 m)	−0.85	−0.06	−0.05	−0.95	−0.81
Sensor 1 (144.51 m)	0.27	−0.90	−0.60	0.37	0.14
Sensor 5 (144.51 m)	−1.00	−0.02	−0.03	−1.00	−0.89
Sensor 5 (144.51 m)	−0.19	−0.67	1.00	−0.20	0.19
Sensor 9 (144.51 m)	−0.98	−0.06	−0.01	−0.99	−0.85
Sensor 9 (144.51 m)	−0.08	−1.00	0.60	−0.01	0.28

Table D.5: The identified measured modes (7-11) of the office tower the Delfste Poort (normalized).

Mode	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
1	1.00										
2	0.00	1.00									
3	0.06	0.03	1.00								
4	0.16	0.01	0.06	1.00							
5	0.03	0.42	0.02	0.00	1.00						
6	0.07	0.02	0.62	0.00	0.03	1.00					
7	0.00	0.68	0.07	0.00	0.72	0.07	1.00				
8	0.16	0.00	0.04	0.35	0.01	0.08	0.00	1.00			
9	0.04	0.00	0.03	0.03	0.02	0.51	0.01	0.06	1.00		
10	0.00	0.04	0.00	0.00	0.42	0.05	0.32	0.00	0.06	1.00	
11	0.03	0.05	0.01	0.00	0.34	0.00	0.18	0.03	0.00	0.82	1.00

Table D.6: The MAC of the modes of Tables D.4 and D.5. The MAC values higher then 0.35 are indicated in lightsteelblue.

D.5. Conclusion

With this appendix, the analysis of obtaining the measured modal properties for the New Orleans and the Delftse Poort is provided, including a verification of the obtained mode shapes. For both buildings, a measurement setup consisting of different acceleration sensors at various heights was used, which successfully identified different modes of the buildings.

The MAC (Modal Assurance Criterion) was used to determine if the obtained mode shapes are orthogonal. For the identified modes 1, 2, 4, 5, 7, and 8 of the residential tower New Orleans, and modes 1, 2, 3, and 4 of the office tower the Delftse Poort, a MAC value lower then 35% was found, indicating that the sensor setup was sufficient to properly distinguish the mode shapes of these modes.

For the identified modes 3, 6, 9, and 10 of the residential tower New Orleans, and modes 5, 6, 7, 8, 9, 10, and 11 of the office tower the Delftse Poort, a MAC value equal or higher then 35% was found, indicating that the sensor setup was not sufficient to properly distinguish the mode shapes of these modes. This suggests that more sensors are needed at different heights, particularly for the office tower the Delfste Poort.

E

Influence of parameters

In this chapter, the results of the residential tower New Orleans of the study "Influence Parameters" in x-direction are given.

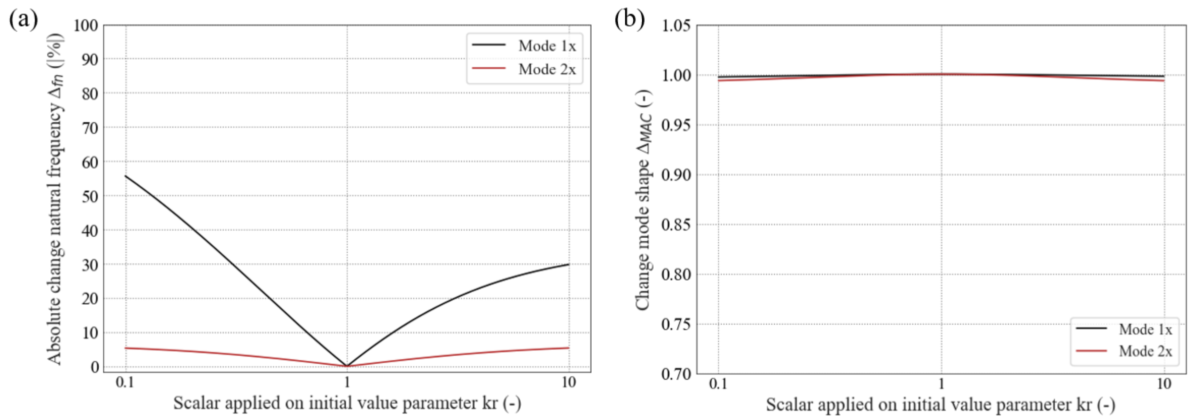

Figure E.1: The influence of parameter K_r on the model output (a,b) in x-direction.

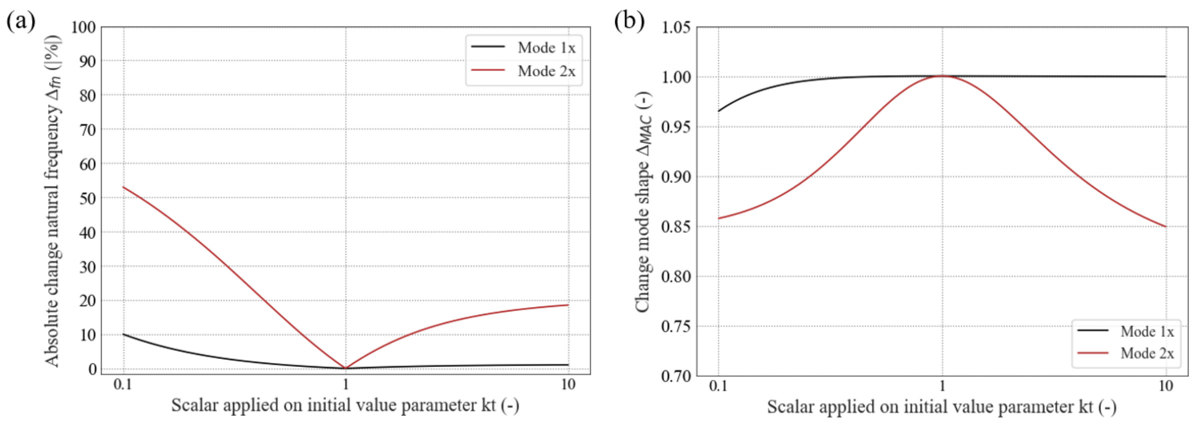

Figure E.2: The influence of parameter K_t on the model output (a,b) in x-direction.

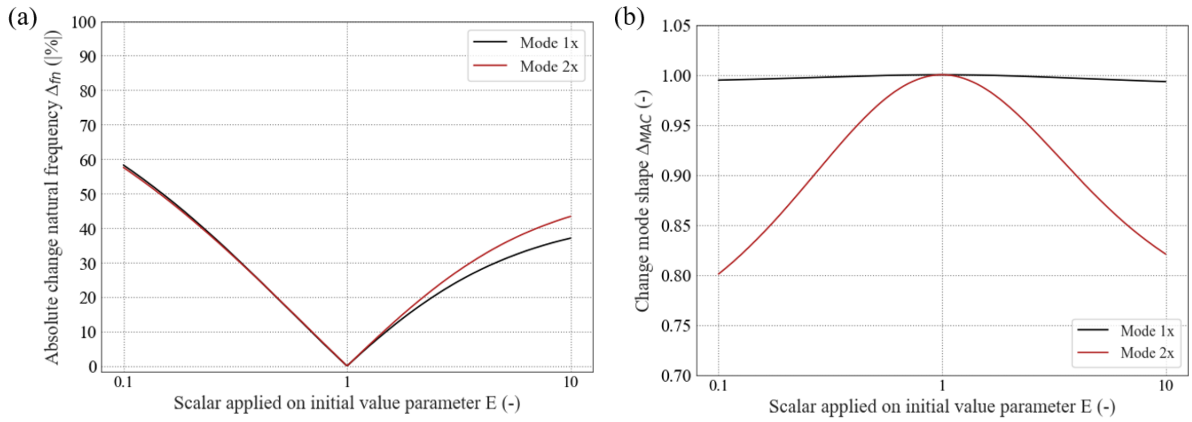

Figure E.3: The influence of parameter E on the model output (a,b) in x-direction.

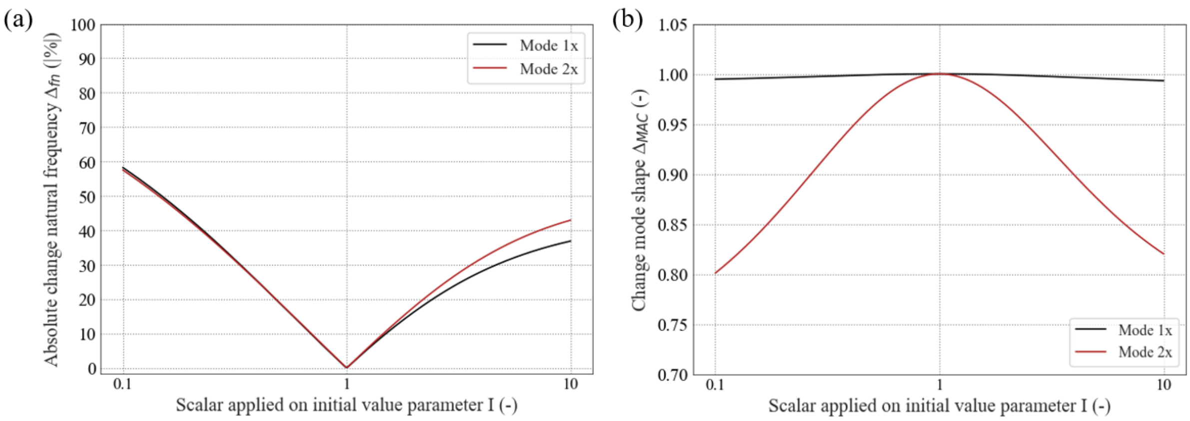

Figure E.4: The influence of parameter I on the model output (a,b) in x-direction.

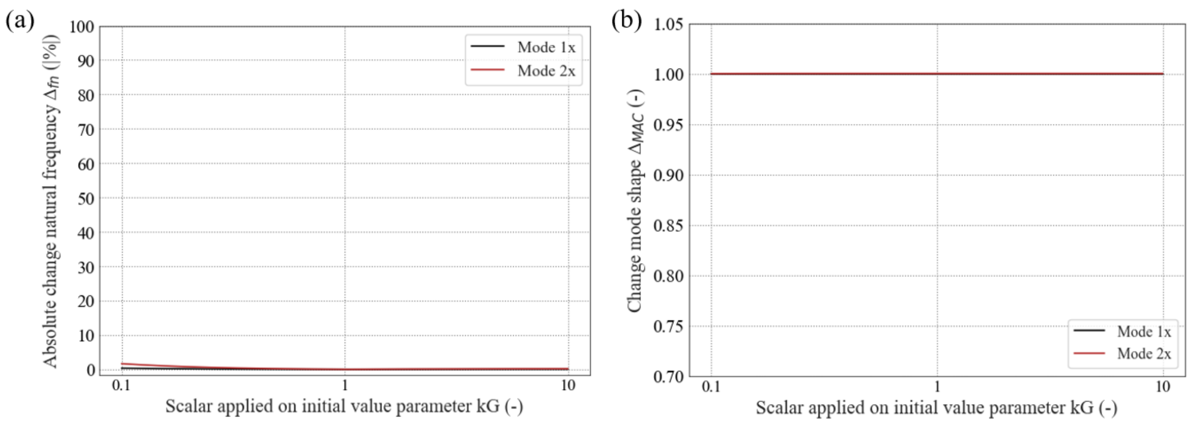

Figure E.5: The influence of parameter kG on the model output (a,b) in x-direction.

F

Python

In this appendix, the Python code used is presented, including the models, model updating procedures, and various studies conducted. Since the studies involve two case studies using the same code, only the code for one case study is shown.

The discrete Timoshenko beam model (model and model updating)

```
from abc import ABC, abstractmethod
from copy import deepcopy
from dataclasses import dataclass
from typing import List, Union, Tuple, Dict, Optional, Any

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import cmath as cm
from scipy.optimize import brentq
from utils.beam_utils import nullspace
from utils.modal_analysis import mac
from functools import partial
from scipy.stats import uniform
from tqdm import tqdm
from skopt import gp_minimize, forest_minimize
from scipy.optimize import Bounds, minimize
from enum import Enum

class UpdatingMethod(Enum):
    CMC = "Crude MonteCarlo"
    MCMC = "Markov-chain MonteCarlo"
    GA = "Genetic algorithm"
    DE = "Differential evolution"
    PSO = "Partical swarm optimsation"
    GP = "Gaussian process"


ParamType = Union[dict, list, np.ndarray]
import matplotlib

matplotlib.use("TkAgg")
----------------------------------- Model Code --------------------------------------------
@dataclass
class Beam1D(ABC):
    params: Dict[str, float]
    height: Union[float, int]
    bc: Optional[Tuple[str, str]] = ("fixed", "free")
    bounds: dict = None
    scales: dict = None
    description: str = ""

    @abstractmethod
    def boundaries(
```

```
            self,
            omega: float,
            factor_m: float,
            factor_v: float,
            kr: float,
            p: float,
            q: float,
            z: Union[np.ndarray, float],
            flag: str
    ):
        pass

    @abstractmethod
    def roots(self,
                 omega: float) -> float:
        pass

@dataclass
class Timo(Beam1D):

    def __init__(self, params, height):
        self.params = deepcopy(params)
        self.orig_params = deepcopy(params)
        self.fn: Union[Dict[str, Any], np.ndarray] = dict()
        self.phi: Union[Dict[str, Any], np.ndarray] = dict()
        self.height = height

    def mode_shapes(self):
        pass

    def roots(self, omega: float):
        s2 = (self.params["E"] * self.params["I"]) / (self.params["kG"] *
        self.params["A"] * (self.params["L"] ** 2))
        b2 = (self.params["rho"] * self.params["A"] * (omega ** 2) *
        (self.params["L"] ** 4)) / (self.params["E"] * self.params["I"])
        r2 = (self.params["I"] / self.params["A"]) / self.params["L"] ** 2

        b = b2 * s2 + b2 * r2
        c = (b2 ** 2) * s2 * r2 - b2
        d = (b ** 2) - 4 * c

        m1 = cm.sqrt((-b - np.sqrt(d)) / 2)
        p = abs(m1.imag)

        if omega < np.sqrt((self.params["kG"] * self.params["A"]) /
        (self.params["rho"] * self.params["I"])):
            m3 = np.sqrt((-b + np.sqrt(d)) / 2)
            q = m3

        elif omega == np.sqrt((self.params["kG"] * self.params["A"]) /
        (self.params["rho"] * self.params["I"])):
            q = 0

        else:
            m3 = cm.sqrt((-b + np.sqrt(d)) / 2)
            q = abs(m3.imag)

        return p, q

    def w(self, p, q, omega: float, z: Union[float, np.ndarray], dz: int = 0):
        if omega < np.sqrt((self.params["kG"] * self.params["A"]) /
        (self.params["rho"] * self.params["I"])):
            if dz == 0:
                return np.array([np.sin(p * z), np.cos(p * z),
                np.sinh(q * z), np.cosh(q * z)])
            elif dz == 1:
                return np.array([p * np.cos(p * z), -p * np.sin(p * z),
                q * np.cosh(q * z), q * np.sinh(q * z)])

        elif omega == np.sqrt((self.params["kG"] * self.params["A"]) /
                        (self.params["rho"] * self.params["I"])):
```

```
            if dz == 0:
                return np.array([np.sin(p * z), np.cos(p * z), 0, 1])
            elif dz == 1:
                return np.array([p * np.cos(p * z), -p * np.sin(p * z), 0, 0])

        else:
            if dz == 0:
                return np.array([np.sin(p * z), np.cos(p * z),
                np.sin(q * z), np.cos(q * z)])
            elif dz == 1:
                return np.array([p * np.cos(p * z), -p * np.sin(p * z),
                q * np.cos(q * z), -q * np.sin(q * z)])

    def phi(self, p, q, omega: float, z: Union[float, np.ndarray], dz: int = 0):
        s2 = (self.params["E"] * self.params["I"]) /
        (self.params["kG"] * self.params["A"] * (self.params["L"] ** 2))
        b2 = (self.params["rho"] * self.params["A"] * (omega ** 2) *
        self.params["L"] ** 4)) / (self.params["E"] * self.params["I"])

        if omega < np.sqrt((self.params["kG"] * self.params["A"]) /
        (self.params["rho"] * self.params["I"])):
            l1 = b2 * s2 / p - p
            l2 = -b2 * s2 / p + p
            l3 = b2 * s2 / q + q
            l4 = l3

            if dz == 0:
                return np.array([l2 * np.cos(p * z), l1 * np.sin(p * z),
                l4 * np.cosh(q * z), l3 * np.sinh(q * z)])
            elif dz == 1:
                return np.array([-l2 * p * np.sin(p * z), l1 * p * np.cos(p * z),
                l4 * q * np.sinh(q * z), l3 * q * np.cosh(q * z)])

        elif omega == np.sqrt((self.params["kG"] * self.params["A"]) /
                              (self.params["rho"] * self.params["I"])):
            l1 = b2 * s2 / p - p
            l2 = -b2 * s2 / p + p

            if dz == 0:
                return np.array([l2 * np.cos(p * z), l1 * np.sin(p * z),
                1, b2 * s2 * z])
            elif dz == 1:
                return np.array([-l2 * p * np.sin(p * z), l1 * p * np.cos(p * z),
                0, b2 * s2])

        else:
            l1 = (b2 * s2 / p) - p
            l2 = -(b2 * s2 / p) + p
            l3 = (b2 * s2 / q) - q
            l4 = -(b2 * s2 / q) + q

            if dz == 0:
                return np.array([l2 * np.cos(p * z), l1 * np.sin(p * z),
                 l4 * np.cos(q * z), l3 * np.sin(q * z)])
            elif dz == 1:
                return np.array([-l2 * p * np.sin(p * z), l1 * p * np.cos(p * z),
                -l4 * q * np.sin(q * z), l3 * q * np.cos(q * z)])

    def factor_continuity(self):
        factor_m = 1 / (self.params["E"] * self.params["I"])
        factor_v = 1 / (self.params["kG"] * self.params["A"])
        return factor_m, factor_v

    # Remove this if frequency dependency is not taken into account of this parameter! This is for the article.
    def kr(self, omega):
        Vs = np.sqrt(self.params["Gs"] / self.params["rhos"])
        a0 = (omega * self.params["D"]) / (2 * Vs)

        b = 2.4 - 0.4 / (self.params["B"] / self.params["D"])**3
        d = 0.55 + 0.01 * np.sqrt((self.params["B"] / self.params["D"]) - 1)
```

```

        krd = 1 - (d * a0**2)/(b + a0**2)
        kr = self.params["kr"] * krd
        return kr
    # Remove kr: float and change kr to self.params["kr"] if the frequency dependency of this parameter is not taken into account
    def boundaries(
            self,
            omega: float,
            factor_m: float,
            factor_v: float,
            kr: float,
            p: float,
            q: float,
            z: Union[np.ndarray, float],
            flag: str
    ):
        if flag.lower() == "continuity+" or flag.lower() == "continuity-":
            mm = np.zeros([4, 4])
        else:
            mm = np.zeros([2, 4])

        if flag.lower() == "fixed":
            mm[0, :] = Timo.w(self, omega=omega, p=p, q=q, z=z)
            mm[1, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z)

        elif flag.lower() == "hinged":
            mm[0, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z, dz=1)
            mm[1, :] = Timo.w(self, omega=omega, p=p, q=q, z=z)

        elif flag.lower() == "rot+transl":
            mm[0, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z, dz=1) -
            (kr / (self.params["E"] * self.params["I"]
            / self.params["L"])) *
            Timo.phi(self, omega=omega, p=p, q=q, z=z)
            mm[1, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z) -
            Timo.w(self, omega=omega, p=p, q=q, z=z, dz=1) +
            (self.params["kt"] / ((self.params["A"] * self.params["kG"])
            / self.params["L"])) * Timo.w(self, omega=omega, p=p, q=q, z=z)

        elif flag.lower() == "rot":
            mm[0, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z, dz=1) -
            (kr / (self.params["E"] *
            self.params["I"] / self.params["L"])) *
            Timo.phi(self, omega=omega, p=p, q=q, z=z)
            mm[1, :] = Timo.w(self, omega=omega, p=p, q=q, z=z)

        elif flag.lower() == "transl":
            mm[0, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z)
            mm[1, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z) -
            Timo.w(self, omega=omega, p=p, q=q, z=z, dz=1) +
            (self.params["kt"] / ((self.params["A"] * self.params["kG"]) /
            self.params["L"])) * Timo.w(self, omega=omega, p=p, q=q,z=z)

        elif flag.lower() == "free":
            mm[0, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z, dz=1)
            mm[1, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z) -
            Timo.w(self, omega=omega, p=p, q=q, z=z, dz=1)

        elif flag.lower() == "continuity+":
            fm = factor_m * self.params["E"] * self.params["I"]
            fv = factor_v * self.params["kG"] * self.params["A"]
            mm[0, :] = Timo.w(self, omega=omega, p=p, q=q, z=z)
            mm[1, :] = Timo.w(self, omega=omega, p=p, q=q, z=z, dz=1)
            mm[2, :] = Timo.phi(self, omega=omega, p=p, q=q, z=z, dz=1) * fm
            mm[3, :] = (Timo.phi(self, omega=omega, p=p, q=q, z=z) -
            Timo.w(self, omega=omega, p=p, q=q, z=z, dz=1)) * fv

        elif flag.lower() == "continuity-":
            mm[0, :] = -Timo.w(self, omega=omega, p=p, q=q, z=z)
            mm[1, :] = -Timo.w(self, omega=omega, p=p, q=q, z=z, dz=1)
```

```
            mm[2, :] = -Timo.phi(self, omega=omega, p=p, q=q, z=z, dz=1)
            mm[3, :] = -(Timo.phi(self, omega=omega, p=p, q=q, z=z) -
            Timo.w(self, omega=omega, p=p, q=q, z=z, dz=1))
        return mm


@dataclass
class TimoPieceWiseBeam1D:
    beams: List[Beam1D]
    bc: tuple
    beam = Timo

    def __init__(self, beams, bc):
        self.beams = beams
        self.params = {k: beam.params[k] for beam in beams for k in beam.params}
        self.bc = bc
        self.orig_params = deepcopy(self.params)

    def update_dict(self, attr: str, new_dict: dict):
        for ix, beam in enumerate(self.beams):
            getattr(self, attr).update(new_dict)

    def scale_parms(self, scalers: dict):
        for ix, beam in enumerate(self.beams):
            for k, val in scalers.items():
                beam.params[k] = beam.orig_params[k] * val

    def update_params(self, updated_params: dict):
        for ix, beam in enumerate(self.beams):
            for k, val in updated_params.items():
                beam.params[k] = val

    def update_originals(self, updated_params):
        for k, val in updated_params.items():
            self.orig_params[k] = val

    def set_dict(self, attr: str, new_dict: dict):
        setattr(self, attr, new_dict)

    def modeshapes(
            self,
            min_fn: float = 0.002,
            max_fn: float = 100,
            pts: Union[np.ndarray, int] = 500,
            n_intervals: int = 100,
            n_modes: int = 3):

        n_b = len(self.beams)
        heights = np.cumsum([0] + [beam.height for beam in self.beams])
        min_omega, max_omega = min_fn * 2 * np.pi, max_fn * 2 * np.pi
        omega_vec = np.logspace(np.log10(min_omega), np.log10(max_omega), n_intervals)
        omega_critical = np.sqrt((self.params["kG"] * self.params["A"]) /
        (self.params["rho"] * self.params["I"]))
        print("The␣critical␣frequency␣of␣the␣Timoshenko␣beam␣model␣is:")
        print(omega_critical/ (2 * np.pi))
        omega_part_1 = omega_vec[omega_vec < omega_critical]
        omega_part_3 = omega_vec[omega_vec > omega_critical]

        omega_part_1 = np.append(omega_part_1, omega_critical - 0.00000001)
        omega_part_3 = np.insert(omega_part_3, 0, omega_critical + 0.00000001)

        fun_v_1 = np.array([self.determinant(omega=om_ii, z=heights) for om_ii in
            omega_part_1])
        fun_v_3 = np.array([self.determinant(omega=om_iii, z=heights) for om_iii in
            omega_part_3])

        idx_change_1 = np.where(np.diff(np.sign(fun_v_1)) != 0)[0]
        nsol = len(idx_change_1)

        if nsol < n_modes:
            fun_v_2 = self.determinant(omega=omega_critical, z=heights)
```

```
            if fun_v_2 == 0:
                nsol += 1

        idx_change_3 = np.where(np.diff(np.sign(fun_v_3)) != 0)[0]
        nsol += len(idx_change_3)

        omega_sol = np.empty(0)

        for ii in idx_change_1:
            if len(omega_sol) == n_modes:
                break
            sol = brentq(self.determinant, omega_part_1[ii], omega_part_1[ii + 1], args=(heights,))
            omega_sol = np.append(omega_sol, sol)

        if len(omega_sol) != n_modes and fun_v_2 == 0:
            omega_sol = np.append(omega_sol, omega_critical)

        for iii in idx_change_3:
            if len(omega_sol) == n_modes:
                break
            sol = brentq(self.determinant, omega_part_3[iii], omega_part_3[iii + 1], args=(heights,))
            omega_sol = np.append(omega_sol, sol)

        if len(omega_sol) < n_modes:
            print("Solution are less than required number of modes. Try a larger interval or larger bounds.")

        if len(omega_sol) == 0:
            print("Failed in obtaining solutions!!")

        omega = omega_sol
        fn = omega / (2 * np.pi)

        if isinstance(pts, int):
            pts_pieces = np.linspace(heights[:-1], heights[1:], pts, endpoint=False).T
            pts = np.insert(pts_pieces.ravel(), len(pts_pieces.ravel()), heights[-1])

        phi = np.zeros((len(pts), len(omega)))
        pts = pts.flatten()

        coeff = np.zeros((n_b * 4, 1))

        for aa, om_aa in enumerate(omega):
            mm = self.determinant(omega=om_aa, z=heights, build=True)
            rtol = 0.1
            not_converge = True

            while not_converge:
                coeff = nullspace(mm, atol=0, rtol=rtol).reshape(-1, 1)

                if coeff.shape == (0, 1) or coeff.shape[0] > 4 * n_b:
                    rtol /= 10
                else:
                    not_converge = False

            for ix, beam in enumerate(self.beams):
                idx_z = (pts >= heights[ix]) & (pts <= heights[ix + 1])

                p_, q_ = beam.roots(omega=om_aa)

                z_ix = pts[idx_z]

                if om_aa != omega_critical:
                    phi[idx_z, aa] = np.sum(
                        np.real(beam.w(p=p_, q=q_, omega=om_aa, z=z_ix) * coeff[4 * ix:(4 * ix + 4)]), axis=0)

                if om_aa == omega_critical and np.linalg.matrix_rank(mm) < 3:
                    print(
```

```
                        'Warning, there is a double eigenvalue, please check with plots if 
                            the procedure goes as intended!')

        phi /= phi[np.argmax(np.abs(phi), axis=0), np.arange(phi.shape[1])]
        self.fn = fn
        self.phi = phi
        return fn, phi, pts

    def determinant(
            self,
            omega: float,
            z: np.ndarray,
            build: bool = False,
    ):
        n_b = len(self.beams)
        mm = np.zeros((4 * n_b, 4 * n_b))
        bc_s, bc_e = self.bc
        f__v = []
        f__m = []
        for ix, beam in enumerate(self.beams):
            fv = beam.factor_continuity()[1]
            fm = beam.factor_continuity()[0]
            f__v.append(fv)
            f__m.append(fm)
            if ix == 0:
                f_m = f__m[0]
                f_v = f__v[0]
            else:
                f_m = f__m[ix - 1]
                f_v = f__v[ix - 1]

            bc_s_ix = (bc_s if ix == 0 else "continuity+")
            bc_e_ix = (bc_e if ix == (n_b - 1) else "continuity-")

            _p = beam.roots(omega=omega)[0]
            _q = beam.roots(omega=omega)[1]

            _kr = beam.kr(omega=omega)

            _mm_s = beam.boundaries(omega=omega, factor_m=f_m, factor_v=f_v, kr=_kr, p=_p, q=
                _q, z=z[ix], flag=bc_s_ix)
            _mm_e = beam.boundaries(omega=omega, factor_m=f_m, factor_v=f_v, kr=_kr, p=_p, q=
                _q, z=z[ix + 1], flag=bc_e_ix)

            c_ix = np.arange(4 * ix, (4 * ix + 4))
            r_st = (ix if ix == 0 else 4 * ix - 2)

            _mm = np.vstack([_mm_s, _mm_e])
            r_ix = np.arange(r_st, r_st + _mm.shape[0])
            mm[np.ix_(r_ix, c_ix)] = _mm

        if build:
            return mm
        else:
            return np.linalg.det(mm)


@dataclass
class TimoPieceWiseBeam2D:
    beams_x: List[Beam1D]
    beams_y: List[Beam1D]
    bc: tuple

    def __init__(self, beams_x, beams_y, bc):
        self.beams_x = beams_x
        self.beams_y = beams_y
        self.params_x = {k: beam.params[k] for beam in beams_x for k in beam.params}
        self.params_y = {k: beam.params[k] for beam in beams_y for k in beam.params}
        self.bc = bc
        self.orig_params_x = deepcopy(self.params_x)
        self.orig_params_y = deepcopy(self.params_y)
```

```

    def update_dict(self, attr: str, new_dict_x: dict, new_dict_y: dict):
        for beam in self.beams_x:
            getattr(beam, attr).update(new_dict_x)
        for beam in self.beams_y:
            getattr(beam, attr).update(new_dict_y)

    def scale_params(self, scalers: dict):
        for k, val in scalers.items():
            for beam in self.beams_x:
                beam.params[k] = beam.orig_params[k] * val
            for beam in self.beams_y:
                beam.params[k] = beam.orig_params[k] * val

    def update_params(self, updated_params_x: dict, updated_params_y: dict):
        for k, val in updated_params_x.items():
            for beam in self.beams_x:
                beam.params[k] = val
        for k, val in updated_params_y.items():
            for beam in self.beams_y:
                beam.params[k] = val

    def update_originals(self, updated_params_x, updated_params_y):
        for k, val in updated_params_x.items():
            self.orig_params_x[k] = val
        for k, val in updated_params_y.items():
            self.orig_params_y[k] = val

    def set_dict(self, attr: str, new_dict_x: dict, new_dict_y: dict):
        for beam in self.beams_x:
            setattr(beam, attr, new_dict_x)
        for beam in self.beams_y:
            setattr(beam, attr, new_dict_y)

    def modeshapes(
            self,
            min_fn: float = 0.002,
            max_fn: float = 40000,
            pts: Union[np.ndarray, int] = 500,
            n_intervals: int = 10000,
            n_modes_x: int = 3,
            n_modes_y: int = 3,
    ):
        fn_x, phi_x, pts_x = TimoPieceWiseBeam1D(self.beams_x, self.bc).modeshapes(min_fn,
            max_fn, pts, n_intervals, n_modes_x)
        fn_y, phi_y, pts_y = TimoPieceWiseBeam1D(self.beams_y, self.bc).modeshapes(min_fn,
            max_fn, pts, n_intervals, n_modes_y)

        self.fn_x = fn_x
        self.phi_x = phi_x
        self.fn_y = fn_y
        self.phi_y = phi_y

        return fn_x, fn_y, phi_x, phi_y, pts_x, pts_y

------------------------------ Model Updating Code ----------------------------------------

def sample_start_points(bounds: ParamType, npts: int, seed: int = 42):
    np.random.seed(seed=seed)
    st_pts = np.zeros((npts, len(bounds)))
    for ii, _bb in enumerate(bounds):
        st_pts[:, ii] = uniform.rvs(_bb[0], _bb[1], size=npts)
    return st_pts


def assemble_boundaries(
    parameters_to_update: ParamType,
    a: float = 0.1,
    b: float = 10,
    logscale: bool = False,
):
```

```python

    if isinstance(parameters_to_update, (list, np.ndarray)):
        bounds = [(a, b)] * len(parameters_to_update)

    elif isinstance(parameters_to_update, dict):
        bounds = []
        for pp, bb in parameters_to_update.items():
            if bb is None:
                bounds.append((a, b))
            else:
                bounds.append((bb[0], bb[1]))
    else:
        raise TypeError

    if logscale:
        bounds = [(np.log(a), np.log(b)) for (a, b) in bounds]
    return bounds


def adjust_modes(phi_hat, phi):
    rmse_plus = np.sqrt(np.mean((phi_hat - phi) ** 2, axis=0))
    rmse_minus = np.sqrt(np.mean((phi_hat + phi) ** 2, axis=0))

    flip = np.ones(len(rmse_plus))
    flip[rmse_plus > rmse_minus] = -1
    phi_flip = phi_hat * flip
    return phi_flip

def cost_func1D(
    theta,
    model,
    y_hat,
    parameters_to_update,
    logscale,
    regularisation: float = None,
    symmetry: float=None
):
    fn = y_hat["fn"]
    n_modes = len(fn)

    if logscale:
        theta = np.array([np.exp(aa) for aa in theta])

    if isinstance(parameters_to_update, (np.ndarray, list)):
        parms_names = parameters_to_update
    elif isinstance(parameters_to_update, dict):
        parms_names = list(parameters_to_update.keys())
    else:
        raise TypeError
    new_parms = {k: th for k, th in zip(parms_names, theta)}
    model.scale_parms(scalers=new_parms)
    model.modeshapes(n_modes=n_modes, pts=y_hat["z"])
    cost_fn = np.sum(np.abs(fn - model.fn) / model.fn)

    if fn.shape != model.fn.shape:
        print("Error in shape of modes (x)")

    if regularisation is not None:
        current_params = np.array([model.params[vv] for vv in parms_names])
        orig_params = np.array([model.orig_params[vv] for vv in parms_names])
        delta_props = ((current_params - orig_params) / orig_params) ** 2
        cost_fn += regularisation * delta_props.sum()

    if "modes" in list(y_hat.keys()):
        phi = np.atleast_2d(y_hat["modes"])
        phi_hat = adjust_modes(model.phi, phi)
        mac_ph_p = np.diag(mac(phi_hat, phi))
        cost_modes = np.sum((1 - mac_ph_p))
        return cost_fn + cost_modes
    else:
        return cost_fn
```

```

def  cost_func2D(
    theta,
    model: TimoPieceWiseBeam2D,
    y_hat,
    parameters_to_update,
    logscale,
    regularisation: float = None,
    symmetry: float = None,
):
    fn_x = y_hat["fn_x"]
    fn_y = y_hat["fn_y"]
    n_modes_x = len(fn_x)
    n_modes_y = len(fn_y)

    if logscale:
        theta = np.array([np.exp(aa) for aa in theta])

    if isinstance(parameters_to_update, (np.ndarray, list)):
        parms_names = parameters_to_update
    elif isinstance(parameters_to_update, dict):
        parms_names = list(parameters_to_update.keys())
    else:
        raise TypeError
    new_parms = {k: th for k, th in zip(parms_names, theta)}
    model.scale_params(scalers=new_parms)
    model.modeshapes(n_modes_x= n_modes_x, n_modes_y=n_modes_y, pts=y_hat["z"])

    cost_fn = np.sum(np.abs(fn_x - model.fn_x) / model.fn_x)
    if fn_x.shape != model.fn_x.shape:
        print("Error␣in␣shape␣of␣modes␣(x)")
    cost_fn += np.sum(np.abs(fn_y - model.fn_y) / model.fn_y)
    if fn_y.shape != model.fn_y.shape:
        print("Error␣in␣shape␣of␣modes␣(y)")

    if regularisation is not None:
        current_params = np.array([model.params[vv] for vv in parms_names])
        orig_params = np.array([model.orig_params[vv] for vv in parms_names])
        delta_props = ((current_params - orig_params) / orig_params) ** 2
        cost_fn += regularisation * delta_props.sum()

    if symmetry is not None:
        npr = len(theta) // 2
        ratios = np.array([theta[ii]/theta[npr+ii] for ii in range(npr)])
        cost_fn += symmetry * np.abs(np.log(ratios)).sum()

    if "modes_x" in list(y_hat.keys()):
        phi_x = np.atleast_2d(y_hat["modes_x"])
        phi_hat_x = adjust_modes(model.phi_x, phi_x)
        mac_ph_p_x = np.diag(mac(phi_hat_x, phi_x))
        cost_modes_x = np.sum((1 - mac_ph_p_x))
        cost_fn += cost_modes_x

    if "modes_y" in list(y_hat.keys()):
        phi_y = np.atleast_2d(y_hat["modes_y"])
        phi_hat_y = adjust_modes(model.phi_y, phi_y)
        mac_ph_p_y = np.diag(mac(phi_hat_y, phi_y))
        cost_modes_y = np.sum((1 - mac_ph_p_y))
        cost_fn += cost_modes_y

    return cost_fn

def update_model(
        model: TimoPieceWiseBeam1D,
        parameters_to_update: ParamType,
        cost_func,
        y_hat: dict,
        npts: int = 100,
        sim_type: UpdatingMethod = UpdatingMethod.CMC,
        optim_method: str = "SLSQP",
        seed: int = 42,
```

```
        verbose: bool = False,
        logscale: bool = False,
        regularisation: float = None,
        symmetry: float = None,
        kwargs_optimizer: dict = None,
):
    if kwargs_optimizer is None:
        kwargs_optimizer = {}

    bounds = assemble_boundaries(
        parameters_to_update=parameters_to_update, logscale=logscale
    )

    xl = [bb[0] for bb in bounds]
    xu = [bb[1] for bb in bounds]

    if isinstance(parameters_to_update, (np.ndarray, list)):
        parms_names = parameters_to_update
    elif isinstance(parameters_to_update, dict):
        parms_names = list(parameters_to_update.keys())

    params_opt = np.array([prm + "_opt" for prm in parms_names])
    params_st = np.array([prm + "_st" for prm in parms_names])

    args_cf = (model, y_hat, parameters_to_update, logscale, regularisation)

    solutions = []
    convergence = []

    objective = partial(
        cost_func,
        model=model,
        y_hat=y_hat,
        logscale=logscale,
        parameters_to_update=parameters_to_update,
        regularisation=regularisation,
        symmetry=symmetry
    )

    if sim_type == UpdatingMethod.CMC:

        st_pts = sample_start_points(bounds=bounds, npts=npts, seed=seed)
        iterations_per_start_point = []
        number_function_evaluations = []

        for ii in tqdm(range(npts)):
            min_kwargs = dict(
                x0=st_pts[ii, :], method=optim_method, bounds=bounds
            )

            sol = minimize(objective, **min_kwargs)
            solutions.append(sol.x)
            convergence.append(sol.fun)
            iterations_per_start_point.append(sol.nit)
            number_function_evaluations.append(sol.nfev)

        #print(iterations_per_start_point)
        #print(number_function_evaluations)

    elif sim_type == UpdatingMethod.GP:

        sol = gp_minimize(
            objective,
            bounds,
            initial_point_generator="lhs",
            n_initial_points=npts,
            n_calls=30,
            verbose=verbose,
        )
        solutions.append(sol.x)
        convergence.append(sol.fun)
```

```
    elif sim_type in [UpdatingMethod.DE, UpdatingMethod.GA, UpdatingMethod.PSO]:
        from pymoo.core.problem import ElementwiseProblem
        import pymoo.optimize as opt
        import importlib

        algorithm_class = getattr(
            importlib.import_module(
                f"pymoo.algorithms.soo.nonconvex.{sim_type.name.lower()}"
                # Here is the difference made between updating methods
            ),
            sim_type.name,
        )
        n_var = len(parameters_to_update)

        class MoProblem(ElementwiseProblem):
            def __init__(self):
                super().__init__(n_var=n_var, n_obj=1, xl=xl, xu=xu)

            def _evaluate(self, x, out, *args, **kwargs):
                out["F"] = objective(x)

        problem = MoProblem()
        algorithm = algorithm_class(pop_size=npts, **kwargs_optimizer)
        res = opt.minimize(problem, algorithm, seed=seed, verbose=verbose)
        solutions = [ind.x for ind in res.pop]
        convergence = [ind.f for ind in res.pop]
        st_pts = np.ones((npts, n_var))

    sols = np.vstack(solutions)
    convergence = np.array(convergence).reshape(-1, 1)

    df_data = np.concatenate([convergence, st_pts, sols], axis=1)
    cols = np.concatenate([np.array(["Convergence"]), params_st, params_opt])
    df_opt = pd.DataFrame(data=df_data, columns=cols)

    return df_opt
```

The discrete Euler-Bernoulli beam model (model only)

```
from abc import ABC, abstractmethod
from copy import deepcopy
from dataclasses import dataclass
from typing import List, Union, Tuple, Dict, Optional

import numpy as np
from scipy.optimize import brentq

from utils.beam_utils import nullspace


def w_z(beta, z: Union[float, np.ndarray], dz: int = 0):
    if dz == 0:
        return np.array(
            [np.cosh(beta * z), np.sinh(beta * z), np.cos(beta * z), np.sin(beta * z)]
        )
    elif dz == 1:
        return beta * np.array(
            [np.sinh(beta * z), np.cosh(beta * z), -np.sin(beta * z), np.cos(beta * z)]
        )
    elif dz == 2:
        return beta ** 2 * np.array(
            [np.cosh(beta * z), np.sinh(beta * z), -np.cos(beta * z), -np.sin(beta * z)]
        )
    elif dz == 3:
        return beta ** 3 * np.array(
            [np.sinh(beta * z), np.cosh(beta * z), np.sin(beta * z), -np.cos(beta * z)]
        )
```

```
@dataclass
class Beam1D(ABC):
    params: Dict[str, float]
    height: Union[float, int]
    bc: Optional[Tuple[str, str]] = ("fixed", "free")

    @abstractmethod
    def evaluate_bc(
            self,
            beta: float,
            factor: float,
            z: Union[np.ndarray, float],
            flag: str
    ):
        pass

    @abstractmethod
    def get_beta(self,
                 omega: float) -> float:
        pass


@dataclass
class BernulliBeam(Beam1D):

    def mode_shapes(self):
        pass

    def determinant_boundary_cond(
            self,
            beta: float,
            z_e: float,
            z_s: float = 0.,
            build: bool = False,
    ):
        mm = np.zeros((4, 4))

        bc_s, bc_e = self.bc

        mm[:2, :] = self.evaluate_bc(beta=beta, z=z_s, flag=bc_s)
        mm[2:, :] = self.evaluate_bc(beta=beta, z=z_e, flag=bc_e)

        mm /= np.max(np.abs(mm), axis=1).reshape(-1, 1)

        if build:
            return mm
        else:
            return np.linalg.det(mm)

    def get_beta(self, omega: float):
        return np.real((omega ** 2 * self.params["rho"] * self.params["A"] *
        self.params["L"]**4) / self.params["ei"]) ** (1 / 4)

    def factor_continuity(self):
        factor = 1 / (self.params["ei"])
        return factor

    def evaluate_bc(
            self,
            beta: float,
            factor: float,
            z: Union[np.ndarray, float],
            flag: str
    ):
        if flag.lower() == "continuity+" or flag.lower() == "continuity-":
            mm = np.zeros([4, 4])
        else:
            mm = np.zeros([2, 4])
        if flag.lower() == "fixed":
            mm[0, :] = w_z(beta, z=z)
            mm[1, :] = w_z(beta, z=z, dz=1)
```

```
        elif flag.lower() == "hinged":
            mm[0, :] = w_z(beta, z=z)
            mm[1, :] = w_z(beta, z=z, dz=2)
        elif flag.lower() == "rot+transl":
            mm[0, :] = w_z(beta, z=z, dz=2) - ((self.params["kr"] *
            self.params["L"]) / self.params["ei"]) *
            w_z(beta, z=z, dz=1)
            mm[1, :] = w_z(beta, z=z, dz=3) + (self.params["kt"] *
            self.params["L"]**3 / self.params["ei"]) *
            w_z(beta, z=z)
        elif flag.lower() == "rot":
            mm[0, :] = w_z(beta, z=z, dz=2) -
            (self.params["kr"] /
            (self.params["ei"] / self.params["L"])) *
            w_z(beta, z=z, dz=1)
            mm[1, :] = w_z(beta, z=z)
        elif flag.lower() == "transl":
            mm[0, :] = w_z(beta, z=z, dz=1)
            mm[1, :] = w_z(beta, z=z, dz=3) +
            (self.params["kt"] * self.params["L"]**3 /
            self.params["ei"]) * w_z(beta, z=z)
        elif flag.lower() == "free":
            mm[0, :] = w_z(beta, z=z, dz=2)
            mm[1, :] = w_z(beta, z=z, dz=3)

        elif flag.lower() == "continuity+":
            mm[0, :] = w_z(beta, z=z)
            mm[1, :] = w_z(beta, z=z, dz=1)
            mm[2, :] = w_z(beta, z=z, dz=2)
            mm[3, :] = w_z(beta, z=z, dz=3)
        elif flag.lower() == "continuity-":
            mm[0, :] = -w_z(beta, z=z)
            mm[1, :] = -w_z(beta, z=z, dz=1)
            mm[2, :] = -w_z(beta, z=z, dz=2) * self.params["ei"] * factor
            mm[3, :] = -w_z(beta, z=z, dz=3) * self.params["ei"] * factor
        return mm


@dataclass
class Model:
    def update_dict(self, attr: str, new_dict: dict):
        for k, val in new_dict.items():
            getattr(self, attr).update[k] = val

    def set_dict(self, attr: str, new_dict: dict):
        setattr(self, attr, new_dict)

    def update_model(self):
        pass


@dataclass
class PieceWiseBeam1D:
    """ """

    beams: List[Beam1D]
    bc: tuple

    def initialize_beams(self):

        beam = BernulliBeam

    def mode_shapes(
            self,
            min_fn: float = 0.002,
            max_fn: float = 100,
            pts: Union[np.ndarray, int] = 100,
            n_intervals: int = 100,
            n_modes: int = 3
    ):
        n_b = len(self.beams)
```

```
        heights = np.cumsum([0] + [beam.height for beam in self.beams])
        min_omega, max_omega = (min_fn * 2 * np.pi, max_fn * 2 * np.pi)
        omega_vec = np.logspace(np.log10(min_omega), np.log10(max_omega), n_intervals)
        fun_v = np.zeros(omega_vec.shape)
        nsol = 0
        idx_change = []
        for ii, om_ii in enumerate(omega_vec):
            fun_v[ii] = self.determinant_boundary_cond(omega=om_ii, z=heights)
            if ii > 0 and np.sign(fun_v[ii]) != np.sign(fun_v[ii - 1]):
                nsol += 1
                idx_change.append(ii - 1)
            if nsol == n_modes:
                break

        omega_sol = []

        for ii in idx_change:
            if len(omega_sol) == n_modes:
                break
            sol = brentq(
                f=self.determinant_boundary_cond, a=omega_vec[ii], b=omega_vec[ii + 1],
                args=(heights)
            )
            omega_sol.append(sol)

        if len(omega_sol) < n_modes:
            print(
                "Solution␣are␣less␣than␣required␣number␣of␣modes."
                "␣Try␣a␣larger␣interval␣or␣larger␣bounds."
            )

        if len(omega_sol) == 0:
            print("Failed␣in␣obtaining␣solutions!!")

        omega = np.array(omega_sol)
        fn = omega / (2 * np.pi)

        if isinstance(pts, int):
            pts_pieces = np.linspace(heights[:-1], heights[1:], pts, endpoint=False).T
            pts = pts_pieces.ravel()
            pts = np.insert(pts, len(pts), heights[-1])

        phi = np.zeros((len(pts), len(omega)))
        pts = np.asarray(pts)
        pts = pts.flatten()

        coeff = np.zeros(((n_b * 4), 1))

        for ii, om_ii in enumerate(omega):
            mm = self.determinant_boundary_cond(
                omega=om_ii, z=heights, build=True
            )
            rtol = 0.1
            not_converge = True
            while not_converge:
                coeff = nullspace(mm, atol=0, rtol=rtol).reshape(-1, 1)
                if coeff.shape == (0, 1) or coeff.shape[0] > 4 * len(self.beams):
                    rtol /= 10
                else:
                    not_converge = False

            for ix, beam in enumerate(self.beams):
                idx_z = (pts >= heights[ix]) & (pts <= heights[ix + 1])

                beta_ = beam.get_beta(omega=om_ii)
                z_ix = pts[idx_z]
                phi[idx_z, ii] = np.sum(np.real(
                    w_z(beta_, z=z_ix) * coeff[4 * ix:(4 * ix + 4)]), axis=0
                )
                phi_ix = phi[idx_z, ii]
```

```

        phi /= phi[np.argmax(np.abs(phi), axis=0), np.arange(phi.shape[1])]

        self.fn = fn
        self.phi = phi

        return fn, phi, pts

    def determinant_boundary_cond(
            self,
            omega: float,
            z: np.ndarray,
            build: bool = False,
    ):
        n_b = len(self.beams)
        mm = np.zeros((4 * n_b, 4 * n_b))

        bc_s, bc_e = self.bc

        f__ = []
        for ix, beam in enumerate(self.beams):
            f = beam.factor_continuity()
            f__.append(f)
            if ix == 0:
                f_ = f__[ix]
            else:
                f_ = f__[ix - 1]
            bc_s_ix = (bc_s if ix == 0 else "continuity+")
            bc_e_ix = (bc_e if ix == (n_b - 1) else "continuity-")

            _beta = beam.get_beta(omega=omega)
            _mm_s = beam.evaluate_bc(beta=_beta, factor=f_, z=z[ix], flag=bc_s_ix)
            _mm_e = beam.evaluate_bc(beta=_beta, factor=f_, z=z[ix + 1], flag=bc_e_ix)

            c_ix = np.arange(4 * ix, (4 * ix + 4))
            r_st = (ix if ix == 0 else 4 * ix - 2)

            _mm = np.vstack([_mm_s, _mm_e])
            r_ix = np.arange(r_st, r_st + _mm.shape[0])
            mm[np.ix_(r_ix, c_ix)] = _mm

        mm /= np.max(np.abs(mm), axis=1).reshape(-1, 1)

        if build:
            return mm
        else:
            return np.linalg.det(mm)
```

Replication of the article by Taciroglu et al. [2]

```
%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko_verification_article import Timo, TimoPieceWiseBeam1D
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Verification study Timoshenko beam model
In this notebook, a verification study is conducted to check the Timoshenko code written. The
     article "Efficient␣model␣updating␣of␣a␣multi-storey␣frame␣and␣its␣foundation␣stiffness␣
    from␣earthquake␣records␣using␣a␣timoshenko␣beam␣model" is replicated.

## Parameters

# Dimension Parameters:
L = 44 #[m]
Lt = np.array([11,11,11,11]) #[m]
B = 23 #[m]
D = 21 #[m]
```

```
A = B * D #[m2]
I = 1/12 * B * D**3 #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([400,400,400,400]) #[kg/m3]
E = np.array([6.6e8,6.6e8,6.6e8,6.6e8]) #[N/m2]
G = np.array([2.75e8,2.75e8,2.75e8,2.75e8]) #[N/m2]
k = 0.85 #[-]

# Springs:
kr = 2.048e12 #[Nm/rad]
kt = 1.841e10 #[N/m]

# Boundary conditions:
bc = ("rot+transl", "free")

#Ground propertiesGs
Gs = 2.68e8 #[N/m]
rhos = 2.7e3 #[kg/m3]
modes = 3

## Procedure

timoshenko = []
for ix in range(len(H)):
    timoshenko.append(Timo(params = {"E": E[ix], "I": I, "L": L,
    "rho": rho[ix], "A": A, "kG": k*G[ix], "Gs":Gs, "rhos":rhos,
    "B": B, "D": D,"kr": kr, "kt": kt},height= H[ix]))
pb = TimoPieceWiseBeam1D(beams=timoshenko, bc=bc)
fn, phi, pts = pb.modeshapes(n_modes=modes)

normalization_factors = phi[-1, :]
phi = phi / normalization_factors

## Results

print("The␣first␣three␣natural␣frequencies␣(in␣Hz)␣of␣the␣Timoshenko␣beam␣are:")
for ii in range(3):
    print(f"Mode␣{ii␣+␣1}␣:", fn[ii])

print(fn)

plt.figure(figsize=(4, 6))

# Plot mode shapes
plt.plot(phi[:,0], pts, color='black', label='Mode␣1', linewidth=1.8)
plt.plot(phi[:,1], pts, '--', color='firebrick', label='Mode␣2', linewidth=1.8)
plt.plot(phi[:,2], pts, '-.', color='mediumblue', label='Mode␣3', linewidth=1.8)

# Set limits and labels
plt.xlim(-1.5, 1.5)
plt.ylim(0, 1.1)
plt.xlabel('Normalized␣displacement␣(-)')
plt.ylabel('Normalized␣height␣(-)')

# Add legend and customize grid
plt.legend()
plt.grid(True, which='both', color='grey', linestyle=':', linewidth=1)  # Customize grid
plt.gca().set_axisbelow(True)  # Ensure grid is behind other plot elements

plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('black')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('black')
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('black')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.show()
```

Comparison with an Euler-Bernoulli beam

```
%load_ext autoreload
%autoreload 2
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam1D
import numpy as np
from pyvibe.models.Euler_Bernoulli import PieceWiseBeam1D, BernulliBeam
from pyvibe.models.Shear import Shear1D, Shear

%matplotlib inline
# Dimension Parameters:
L = 44 #[m]
Lt = np.array([11,11,11,11]) #[m]
B = 23 #[m]
D = 21 #[m]
A = B * D #[m2]
I = 1/12 * B * D**3
H = Lt/L

# Material properties:
rho = np.array([400,400,400,400]) #[kg/m3]
E = np.array([6.6e8,6.6e8,6.6e8,6.6e8])  #[N/m2]
G = np.array([2.75e8,2.75e8,2.75e8,2.75e8])  #[N/m2]
k = 0.85 #[-]

# Springs
kr = 2.048e12 #[Nm/rad]
kt = 1.841e10 #[N/m]

# Boundary conditions:
bc = ("rot+transl", "free")

# Scalar boundaries
amount_points = 50
left = 1
right = 100
# MAC
def adjust_modes(phi1, phi2):
    rmse_plus = np.sqrt(np.mean((phi1 - phi2) ** 2))
    rmse_minus = np.sqrt(np.mean((phi1 + phi2) ** 2))
    flip = np.ones_like(phi2)
    flip[rmse_plus > rmse_minus] = -1
    phi_flip = phi1 * flip
    return phi_flip

def modal_assurance_criterion(phi1, phi2):
    phi2 = adjust_modes(phi2, phi1)
    mac = np.abs(np.conj(phi1).dot(phi2))**2 / \
              (np.conj(phi1).dot(phi1) * np.conj(phi2).dot(phi2))
    return mac
timoshenko = []
for ix in range(len(H)):
    timoshenko.append(Timo(params={"E": E[ix], "I": I,"L": L,
    "rho": rho[ix], "A": A , "kG": k * G[ix], "kr": kr, "kt": kt}, height=H[ix]))
model = TimoPieceWiseBeam1D(beams=timoshenko, bc = bc)
fn, phi, pts = model.modeshapes()
change = np.logspace(np.log10(left), np.log10(right), amount_points)
euler = []
for ix in range(len(H)):
    euler.append(BernulliBeam(params={"ei": E[ix]*I, "L": L,
    "rho": rho[ix], "A": A , "kr": kr, "kt": kt}, height=H[ix]))
model = PieceWiseBeam1D(beams=euler, bc = bc)
fn_e, phi_e, pts_e = model.mode_shapes()


def change_kG(change, E, I, L, rho, A, k, G, kr, kt, H,
         bc):
    f1 = []
    f2 = []
    f3 = []
```

```
    m1 = []
    m2 = []
    m3 = []

    for value in change:
        timoshenko = []
        for ix in range(len(H)):
            timoshenko.append(Timo(params={"E": E[ix], "I": I,"L": L,
            "rho": rho[ix], "A": A , "kG": k * G[ix] * value, "kr": kr,
            "kt": kt}, height=H[ix]))
        model = TimoPieceWiseBeam1D(beams=timoshenko, bc = bc)
        fn, phi, pts = model.modeshapes()

        f1.append(fn[0])
        f2.append(fn[1])
        f3.append(fn[2])
        m1.append(modal_assurance_criterion(phi_e[:, 0], phi[:, 0]))
        m2.append(modal_assurance_criterion(phi_e[:, 1], phi[:, 1]))
        m3.append(modal_assurance_criterion(phi_e[:, 2], phi[:, 2]))

    return f1, f2, f3,m1,m2,m3

    f1_kG, f2_kG, f3_kG, m1_kG, m2_kG, m3_kG = change_kG(change, E, I, L, rho, A, k, G, kr, kt, H, bc)

    import numpy as np
import matplotlib.pyplot as plt

def plot_elements(f1_kG, f2_kG, f3_kG, change, fn_e, t):
    plt.figure(figsize=(7, 5))

    fn_e_0_array = np.full_like(change, fn_e[0])
    fn_e_1_array = np.full_like(change, fn_e[1])
    fn_e_2_array = np.full_like(change, fn_e[2])

    plt.plot(change, fn_e_0_array, color='black', linestyle='dotted', linewidth=1.3, zorder=1)
    plt.plot(change, fn_e_1_array, color='black', linestyle='dotted', linewidth=1.3, zorder=1)
    plt.plot(change, fn_e_2_array, color='black', linestyle='dotted', linewidth=1.3, zorder=1)

    plt.text(change[0], fn_e_0_array[0] + 1, f'Euler␣$\omega_1$␣:␣{fn_e[0]:.2f}␣Hz', fontsize=10, color='black', zorder=2, fontname='Arial')
    plt.text(change[0], fn_e_1_array[0] + 1, f'Euler␣$\omega_2$:␣{fn_e[1]:.2f}␣Hz', fontsize=10, color='black', zorder=2, fontname='Arial')
    plt.text(change[0], fn_e_2_array[0] + 1, f'Euler␣$\omega_3$:␣{fn_e[2]:.2f}␣Hz', fontsize=10, color='black', zorder=2, fontname='Arial')

    plt.scatter(change, f1_kG,  color='black', linewidth=1.3,
    label='Timoshenko␣$\omega_{1}$', s=9)
    plt.scatter(change, f2_kG,  color='firebrick', linewidth=1.3,
    label='Timoshenko␣$\omega_{2}$', s=9)
    plt.scatter(change, f3_kG,  color='mediumblue', linewidth=1.3,
    label='Timoshenko␣$\omega_{3}$', s=9)

    plt.xlabel('Scalar␣applied␣on␣the␣initial␣value␣of␣parameter␣$G$␣(-)', fontsize=12, fontname='Arial')
    plt.ylabel('Natural␣frequency␣$f_n$␣(Hz)', fontsize=12, fontname='Arial')
    plt.ylim(-1, 30)
    plt.xlim(0.9, 110)
    plt.gca().set_facecolor('white')
    plt.grid(True, which='both', color='grey', linestyle=':', linewidth=0.5)
    plt.xscale('log')
    #plt.title('Model output Timoshenko and Euler-Bernoulli beam model', fontsize=14, fontname='Arial')
    plt.legend(fontsize=10, prop={'family': 'Arial'})
    plt.gca().spines['top'].set_linewidth(0.5)
    plt.gca().spines['top'].set_color('black')
    plt.gca().spines['right'].set_linewidth(0.5)
    plt.gca().spines['right'].set_color('black')
```

```python
    plt.gca().spines['bottom'].set_linewidth(0.5)
    plt.gca().spines['bottom'].set_color('black')
    plt.gca().spines['left'].set_linewidth(0.5)
    plt.gca().spines['left'].set_color('black')
    plt.gca().set_xticks([1, 10, 100])
    plt.gca().set_xticklabels(['1', '10', '100'], fontname='Arial')
    plt.show()
    plot_elements(f1_kG, f2_kG, f3_kG, change, fn_e, 2)
```

Models

```python
%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam1D, TimoPieceWiseBeam2D, UpdatingMethod, cost_func1D, cost_func2D,update_model
import numpy as np
from pyvibe.models.Euler_Bernoulli import PieceWiseBeam1D, BernulliBeam
import matplotlib.pyplot as plt
%matplotlib inline

## Parameters

# Dimension Parameters:
L = 160.58 #[m]
Lt = np.array([5.735, 11.47, 26.640, 89.91, 26.825]) #[m]
B = 28 #[m]
D = 28 #[m]
A = B * D #[m2]
#I_x = np.array([619.21, 948.47, 2727.22, 2335.82, 905.52]) #[m4]
#I_y = np.array([1536.25, 959.01, 1245.46, 1202.07, 831.57]) #[m4]
I_yy = np.array([1444,1500,1393,1389,1293]) #[m4]
I_xx = np.array([1532,1548,2796,2324,760]) #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([1933, 495, 486, 440, 383]) #[kg/m3]
E = np.array([38.2e9, 38.2e9, 38.2e9, 38.2e9, 32.8e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v)) #[N/m^2]
k = 0.85 #[-]

# Springs:
kr_x = 2380e9 #[Nm/rad]
kr_y = 2625e9 #[Nm/rad]
kt_y = 19e9 #[N/m]
kt_x = 4e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")

## Procedure

timoshenko_x = []
for ix in range(len(H)):
    timoshenko_x.append(Timo(params={"E": E[ix], "I": I_xx[ix],"L": L,"rho": rho[ix], "A": A, "kG": k * G[ix], "kr": kr_x, "kt": kt_x}, height=H[ix]))
timoshenko_y = []
for ix in range(len(H)):
    timoshenko_y.append(Timo(params={"E": E[ix], "I": I_yy[ix],"L": L,"rho": rho[ix], "A": A, "kG": k * G[ix], "kr": kr_y, "kt": kt_y}, height=H[ix]))
model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc = bc)
fn_x, fn_y, phi_x, phi_y, pts_x, pts_y = model2D.modeshapes()

print("The first two natural frequencies (in Hz) in x-direction are:")
for ii in range(2):
    print(f"Mode {ii + 1} :", fn_x[ii])

print("The first three natural frequencies (in Hz) in y-direction are:")
for ii in range(3):
```

```python
    print(f"Mode {ii + 1} :", fn_y[ii])

euler_x = []
for ix in range(len(H)):
    euler_x.append(BernulliBeam(params={"ei": E[ix]*I_xx[ix], "L": L,"rho": rho[ix], "A": A , "kr": kr_x, "kt": kt_x}, height=H[ix]))
model_x = PieceWiseBeam1D(beams=euler_x, bc = bc)
fn_xe, phi_xe, pts_xe = model_x.mode_shapes()
euler_y = []
for ix in range(len(H)):
    euler_y.append(BernulliBeam(params={"ei": E[ix]*I_yy[ix], "L": L,"rho": rho[ix], "A": A , "kr": kr_y, "kt": kt_y}, height=H[ix]))
model_y = PieceWiseBeam1D(beams=euler_y, bc = bc)
fn_ye, phi_yie, pts_ye = model_y.mode_shapes()

print("The first two natural frequencies (in Hz) in x-direction are:")
for ii in range(2):
    print(f"Mode {ii + 1} :", fn_xe[ii])

print("The first three natural frequencies (in Hz) in y-direction are:")
for ii in range(3):
    print(f"Mode {ii + 1} :", fn_ye[ii])

# Dimension Parameters:
L = 160.58 #[m]
Lt = np.array([160.58]) #[m]
B = 28 #[m]
D = 28 #[m]
A = B * D #[m2]
I_yy = np.array([1384]) #[m4]
I_xx = np.array([2057]) #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([545]) #[kg/m3]
E = np.array([37.3e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v)) #[N/m^2]
k = 0.85 #[-]

# Springs:
kr_x = 2380e9 #[Nm/rad]
kr_y = 2625e9 #[Nm/rad]
kt_y = 19e9 #[N/m]
kt_x = 4e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")

timoshenko_xu = []
for ix in range(len(H)):
    timoshenko_xu.append(Timo(params={"E": E[ix], "I": I_xx[ix],"L": L,"rho": rho[ix], "A": A , "kG": k * G[ix], "kr": kr_x, "kt": kt_x}, height=H[ix]))
timoshenko_yu = []
for ix in range(len(H)):
    timoshenko_yu.append(Timo(params={"E": E[ix], "I": I_yy[ix],"L": L,"rho": rho[ix], "A": A , "kG": k * G[ix], "kr": kr_y, "kt": kt_y}, height=H[ix]))
model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_xu, beams_y=timoshenko_yu, bc = bc)
fn_xu, fn_yu, phi_xu, phi_yu, pts_xu, pts_yu = model2D.modeshapes()
print(timoshenko_yu)
print("The first two natural frequencies (in Hz) in x-direction are:")
for ii in range(2):
    print(f"Mode {ii + 1} :", fn_xu[ii])

print("The first three natural frequencies (in Hz) in y-direction are:")
for ii in range(3):
    print(f"Mode {ii + 1} :", fn_yu[ii])
```

Input Parameters

```python
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam1D, TimoPieceWiseBeam2D, UpdatingMethod, cost_func1D, cost_func2D,update_model
import numpy as np
import matplotlib.pyplot as plt

# MAC
def adjust_modes(phi1, phi2):
    rmse_plus = np.sqrt(np.mean((phi1 - phi2) ** 2))
    rmse_minus = np.sqrt(np.mean((phi1 + phi2) ** 2))
    flip = np.ones_like(phi2)
    flip[rmse_plus > rmse_minus] = -1
    phi_flip = phi1 * flip
    return phi_flip

def modal_assurance_criterion(phi1, phi2):
    phi2 = adjust_modes(phi2, phi1)
    mac = np.abs(np.conj(phi1).dot(phi2))**2 / \
            (np.conj(phi1).dot(phi1) * np.conj(phi2).dot(phi2))
    return mac
# ------------------------------------------------------------------------------------------------------------------------------
# Parameter kr
def change_kr(change, fn_x, fn_y, phi_x, phi_y, E_x, E_y, I_x, I_y, L, rho, A, k, G_x, G_y, kr_x, kr_y, kt_x, kt_y, H,
            bc):
    Change_f1_x = []
    Change_f2_x = []
    Change_f3_x = []
    Change_f1_y = []
    Change_f2_y = []
    Change_f3_y = []
    Change_m1_x = []
    Change_m2_x = []
    Change_m3_x = []
    Change_m1_y = []
    Change_m2_y = []
    Change_m3_y = []

    for value in change:
        kr_new_x = kr_x * value
        kr_new_y = kr_y * value
        timoshenko_x = []
        for ix in range(len(H)):
            timoshenko_x.append(Timo(
                params={"E": E_x[ix], "I": I_x[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG": k * G_x[ix], "kr": kr_new_x,
                    "kt": kt_x}, height=H[ix]))
        timoshenko_y = []
        for ix in range(len(H)):
            timoshenko_y.append(Timo(
                params={"E": E_y[ix], "I": I_y[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG": k * G_y[ix], "kr": kr_new_y,
                    "kt": kt_y}, height=H[ix]))

        model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc=bc)
        fn_x_kr, fn_y_kr, phi_x_kr, phi_y_kr, pts_x_kr, pts_y_kr = model2D.modeshapes()

        Change_f1_x.append(((fn_x_kr[0] - fn_x[0]) / fn_x[0]) * 100)
        Change_f2_x.append(((fn_x_kr[1] - fn_x[1]) / fn_x[1]) * 100)
        Change_f3_x.append(((fn_x_kr[2] - fn_x[2]) / fn_x[2]) * 100)
        Change_f1_y.append(((fn_y_kr[0] - fn_y[0]) / fn_y[0]) * 100)
        Change_f2_y.append(((fn_y_kr[1] - fn_y[1]) / fn_y[1]) * 100)
        Change_f3_y.append(((fn_y_kr[2] - fn_y[2]) / fn_y[2]) * 100)

        Change_m1_x.append(modal_assurance_criterion(phi_x[:, 0], phi_x_kr[:, 0]))
        Change_m2_x.append(modal_assurance_criterion(phi_x[:, 1], phi_x_kr[:, 1]))
        Change_m3_x.append(modal_assurance_criterion(phi_x[:, 2], phi_x_kr[:, 2]))
```

```
        Change_m1_y.append(modal_assurance_criterion(phi_y[:, 0], phi_y_kr[:, 0]))
        Change_m2_y.append(modal_assurance_criterion(phi_y[:, 1], phi_y_kr[:, 1]))
        Change_m3_y.append(modal_assurance_criterion(phi_y[:, 2], phi_y_kr[:, 2]))
    return Change_f1_x, Change_f2_x, Change_f3_x, Change_f1_y, Change_f2_y, Change_f3_y,
        Change_m1_x, Change_m2_x, Change_m3_x, Change_m1_y, Change_m2_y, Change_m3_y

# Parameter kt
def change_kt(change, fn_x, fn_y, phi_x, phi_y, E_x, E_y, I_x, I_y, L, rho, A, k, G_x, G_y,
    kr_x, kr_y, kt_x, kt_y, H,
              bc):
    Change_f1_x = []
    Change_f2_x = []
    Change_f3_x = []
    Change_f1_y = []
    Change_f2_y = []
    Change_f3_y = []
    Change_m1_x = []
    Change_m2_x = []
    Change_m3_x = []
    Change_m1_y = []
    Change_m2_y = []
    Change_m3_y = []

    for value in change:
        kt_new_x = kt_x * value
        kt_new_y = kt_y * value
        timoshenko_x = []
        for ix in range(len(H)):
            timoshenko_x.append(Timo(
                params={"E": E_x[ix], "I": I_x[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG":
                     k * G_x[ix], "kr": kr_x,
                        "kt": kt_new_x}, height=H[ix]))
        timoshenko_y = []
        for ix in range(len(H)):
            timoshenko_y.append(Timo(
                params={"E": E_y[ix], "I": I_y[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG":
                     k * G_y[ix], "kr": kr_y,
                        "kt": kt_new_y}, height=H[ix]))

        model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc=bc)
        fn_x_kt, fn_y_kt, phi_x_kt, phi_y_kt, pts_x_kt, pts_y_kt = model2D.modeshapes()

        Change_f1_x.append(((fn_x_kt[0] - fn_x[0]) / fn_x[0]) * 100)
        Change_f2_x.append(((fn_x_kt[1] - fn_x[1]) / fn_x[1]) * 100)
        Change_f3_x.append(((fn_x_kt[2] - fn_x[2]) / fn_x[2]) * 100)
        Change_f1_y.append(((fn_y_kt[0] - fn_y[0]) / fn_y[0]) * 100)
        Change_f2_y.append(((fn_y_kt[1] - fn_y[1]) / fn_y[1]) * 100)
        Change_f3_y.append(((fn_y_kt[2] - fn_y[2]) / fn_y[2]) * 100)

        Change_m1_x.append(modal_assurance_criterion(phi_x[:, 0], phi_x_kt[:, 0]))
        Change_m2_x.append(modal_assurance_criterion(phi_x[:, 1], phi_x_kt[:, 1]))
        Change_m3_x.append(modal_assurance_criterion(phi_x[:, 2], phi_x_kt[:, 2]))
        Change_m1_y.append(modal_assurance_criterion(phi_y[:, 0], phi_y_kt[:, 0]))
        Change_m2_y.append(modal_assurance_criterion(phi_y[:, 1], phi_y_kt[:, 1]))
        Change_m3_y.append(modal_assurance_criterion(phi_y[:, 2], phi_y_kt[:, 2]))
    return Change_f1_x, Change_f2_x, Change_f3_x, Change_f1_y, Change_f2_y, Change_f3_y,
        Change_m1_x, Change_m2_x, Change_m3_x, Change_m1_y, Change_m2_y, Change_m3_y

# Parameter I
def change_I(change, fn_x, fn_y, phi_x, phi_y, E_x, E_y, I_x, I_y, L, rho, A, k, G_x, G_y,
    kr_x, kr_y, kt_x, kt_y, H,
             bc):
    Change_f1_x = []
    Change_f2_x = []
    Change_f3_x = []
    Change_f1_y = []
    Change_f2_y = []
    Change_f3_y = []
    Change_m1_x = []
    Change_m2_x = []
    Change_m3_x = []
```

```
    Change_m1_y = []
    Change_m2_y = []
    Change_m3_y = []

    for value in change:
        print(value)
        I_new_x = I_x * value
        I_new_y = I_y * value
        timoshenko_x = []
        for ix in range(len(H)):
            timoshenko_x.append(Timo(
                params={"E": E_x[ix], "I": I_new_x[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG": k * G_x[ix], "kr": kr_x,
                        "kt": kt_x}, height=H[ix]))
        timoshenko_y = []
        for ix in range(len(H)):
            timoshenko_y.append(Timo(
                params={"E": E_y[ix], "I": I_new_y[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG": k * G_y[ix], "kr": kr_y,
                        "kt": kt_y}, height=H[ix]))

        model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc=bc)
        fn_x_I, fn_y_I, phi_x_I, phi_y_I, pts_x_I, pts_y_I = model2D.modeshapes()
        print(fn_x_I)

        Change_f1_x.append(((fn_x_I[0] - fn_x[0]) / fn_x[0]) * 100)
        Change_f2_x.append(((fn_x_I[1] - fn_x[1]) / fn_x[1]) * 100)
        Change_f3_x.append(((fn_x_I[2] - fn_x[2]) / fn_x[2]) * 100)
        Change_f1_y.append(((fn_y_I[0] - fn_y[0]) / fn_y[0]) * 100)
        Change_f2_y.append(((fn_y_I[1] - fn_y[1]) / fn_y[1]) * 100)
        Change_f3_y.append(((fn_y_I[2] - fn_y[2]) / fn_y[2]) * 100)

        Change_m1_x.append(modal_assurance_criterion(phi_x[:, 0], phi_x_I[:, 0]))
        Change_m2_x.append(modal_assurance_criterion(phi_x[:, 1], phi_x_I[:, 1]))
        Change_m3_x.append(modal_assurance_criterion(phi_x[:, 2], phi_x_I[:, 2]))
        Change_m1_y.append(modal_assurance_criterion(phi_y[:, 0], phi_y_I[:, 0]))
        Change_m2_y.append(modal_assurance_criterion(phi_y[:, 1], phi_y_I[:, 1]))
        Change_m3_y.append(modal_assurance_criterion(phi_y[:, 2], phi_y_I[:, 2]))
    return Change_f1_x, Change_f2_x, Change_f3_x, Change_f1_y, Change_f2_y, Change_f3_y, Change_m1_x, Change_m2_x, Change_m3_x, Change_m1_y, Change_m2_y, Change_m3_y

# Parameter E
def change_E(change, fn_x, fn_y, phi_x, phi_y, E_x, E_y, I_x, I_y, L, rho, A, k, G_x, G_y, kr_x, kr_y, kt_x, kt_y, H,
             bc):
    Change_f1_x = []
    Change_f2_x = []
    Change_f3_x = []
    Change_f1_y = []
    Change_f2_y = []
    Change_f3_y = []
    Change_m1_x = []
    Change_m2_x = []
    Change_m3_x = []
    Change_m1_y = []
    Change_m2_y = []
    Change_m3_y = []


    for value in change:
        E_new_x = E_x * value
        E_new_y = E_y * value
        timoshenko_x = []
        for ix in range(len(H)):
            timoshenko_x.append(Timo(
                params={"E": E_new_x[ix], "I": I_x[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG": k * G_x[ix], "kr": kr_x,
                        "kt": kt_x}, height=H[ix]))
        timoshenko_y = []
        for ix in range(len(H)):
            timoshenko_y.append(Timo(
```

```
                params={"E": E_new_y[ix], "I": I_y[ix], "L": L, "rho": rho[ix], "A": A[ix], "
                    kG": k * G_y[ix], "kr": kr_y,
                        "kt": kt_y}, height=H[ix]))

        model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc=bc)
        fn_x_E, fn_y_E, phi_x_E, phi_y_E, pts_x_E, pts_y_E = model2D.modeshapes()

        Change_f1_x.append(((fn_x_E[0] - fn_x[0]) / fn_x[0]) * 100)
        Change_f2_x.append(((fn_x_E[1] - fn_x[1]) / fn_x[1]) * 100)
        Change_f3_x.append(((fn_x_E[2] - fn_x[2]) / fn_x[2]) * 100)
        Change_f1_y.append(((fn_y_E[0] - fn_y[0]) / fn_y[0]) * 100)
        Change_f2_y.append(((fn_y_E[1] - fn_y[1]) / fn_y[1]) * 100)
        Change_f3_y.append(((fn_y_E[2] - fn_y[2]) / fn_y[2]) * 100)

        Change_m1_x.append(modal_assurance_criterion(phi_x[:, 0], phi_x_E[:, 0]))
        Change_m2_x.append(modal_assurance_criterion(phi_x[:, 1], phi_x_E[:, 1]))
        Change_m3_x.append(modal_assurance_criterion(phi_x[:, 2], phi_x_E[:, 2]))
        Change_m1_y.append(modal_assurance_criterion(phi_y[:, 0], phi_y_E[:, 0]))
        Change_m2_y.append(modal_assurance_criterion(phi_y[:, 1], phi_y_E[:, 1]))
        Change_m3_y.append(modal_assurance_criterion(phi_y[:, 2], phi_y_E[:, 2]))
    return Change_f1_x, Change_f2_x, Change_f3_x, Change_f1_y, Change_f2_y, Change_f3_y,
        Change_m1_x, Change_m2_x, Change_m3_x, Change_m1_y, Change_m2_y, Change_m3_y

# Parameter kG
def change_kG(change, fn_x, fn_y, phi_x, phi_y, E_x, E_y, I_x, I_y, L, rho, A, k, G_x, G_y,
    kr_x, kr_y, kt_x, kt_y, H,
              bc):
    Change_f1_x = []
    Change_f2_x = []
    Change_f3_x = []
    Change_f1_y = []
    Change_f2_y = []
    Change_f3_y = []
    Change_m1_x = []
    Change_m2_x = []
    Change_m3_x = []
    Change_m1_y = []
    Change_m2_y = []
    Change_m3_y = []

    for value in change:
        kG_new_x = k * G_x * value
        kG_new_y = k * G_y * value
        timoshenko_x = []
        for ix in range(len(H)):
            timoshenko_x.append(Timo(
                params={"E": E_x[ix], "I": I_x[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG":
                     kG_new_x[ix], "kr": kr_x,
                        "kt": kt_x}, height=H[ix]))
        timoshenko_y = []
        for ix in range(len(H)):
            timoshenko_y.append(Timo(
                params={"E": E_y[ix], "I": I_y[ix], "L": L, "rho": rho[ix], "A": A[ix], "kG":
                     kG_new_y[ix], "kr": kr_y,
                        "kt": kt_y}, height=H[ix]))

        model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc=bc)
        fn_x_kG, fn_y_kG, phi_x_kG, phi_y_kG, pts_x_kG, pts_y_kG = model2D.modeshapes()

        Change_f1_x.append(((fn_x_kG[0] - fn_x[0]) / fn_x[0]) * 100)
        Change_f2_x.append(((fn_x_kG[1] - fn_x[1]) / fn_x[1]) * 100)
        Change_f3_x.append(((fn_x_kG[2] - fn_x[2]) / fn_x[2]) * 100)
        Change_f1_y.append(((fn_y_kG[0] - fn_y[0]) / fn_y[0]) * 100)
        Change_f2_y.append(((fn_y_kG[1] - fn_y[1]) / fn_y[1]) * 100)
        Change_f3_y.append(((fn_y_kG[2] - fn_y[2]) / fn_y[2]) * 100)

        Change_m1_x.append(modal_assurance_criterion(phi_x[:, 0], phi_x_kG[:, 0]))
        Change_m2_x.append(modal_assurance_criterion(phi_x[:, 1], phi_x_kG[:, 1]))
        Change_m3_x.append(modal_assurance_criterion(phi_x[:, 2], phi_x_kG[:, 2]))
        Change_m1_y.append(modal_assurance_criterion(phi_y[:, 0], phi_y_kG[:, 0]))
        Change_m2_y.append(modal_assurance_criterion(phi_y[:, 1], phi_y_kG[:, 1]))
```

```
        Change_m3_y.append(modal_assurance_criterion(phi_y[:, 2], phi_y_kG[:, 2]))
    return Change_f1_x, Change_f2_x, Change_f3_x, Change_f1_y, Change_f2_y, Change_f3_y,
        Change_m1_x, Change_m2_x, Change_m3_x, Change_m1_y, Change_m2_y, Change_m3_y

# Parameter rho
def change_rho(change, fn_x, fn_y, phi_x, phi_y, E_x, E_y, I_x, I_y, L, rho, A, k, G_x, G_y,
    kr_x, kr_y, kt_x, kt_y, H,
               bc):
    Change_f1_x = []
    Change_f2_x = []
    Change_f3_x = []
    Change_f1_y = []
    Change_f2_y = []
    Change_f3_y = []
    Change_m1_x = []
    Change_m2_x = []
    Change_m3_x = []
    Change_m1_y = []
    Change_m2_y = []
    Change_m3_y = []

    for value in change:
        rho_new = rho * value
        timoshenko_x = []
        for ix in range(len(H)):
            timoshenko_x.append(Timo(
                params={"E": E_x[ix], "I": I_x[ix], "L": L, "rho": rho_new[ix], "A": A[ix], "
                    kG": k * G_x[ix], "kr": kr_x,
                        "kt": kt_x}, height=H[ix]))
        timoshenko_y = []
        for ix in range(len(H)):
            timoshenko_y.append(Timo(
                params={"E": E_y[ix], "I": I_y[ix], "L": L, "rho": rho_new[ix], "A": A[ix], "
                    kG": k * G_y[ix], "kr": kr_y,
                        "kt": kt_y}, height=H[ix]))

        model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc=bc)
        fn_x_rho, fn_y_rho, phi_x_rho, phi_y_rho, pts_x_rho, pts_y_rho = model2D.modeshapes()

        Change_f1_x.append(((fn_x_rho[0] - fn_x[0]) / fn_x[0]) * 100)
        Change_f2_x.append(((fn_x_rho[1] - fn_x[1]) / fn_x[1]) * 100)
        Change_f3_x.append(((fn_x_rho[2] - fn_x[2]) / fn_x[2]) * 100)
        Change_f1_y.append(((fn_y_rho[0] - fn_y[0]) / fn_y[0]) * 100)
        Change_f2_y.append(((fn_y_rho[1] - fn_y[1]) / fn_y[1]) * 100)
        Change_f3_y.append(((fn_y_rho[2] - fn_y[2]) / fn_y[2]) * 100)

        Change_m1_x.append(modal_assurance_criterion(phi_x[:, 0], phi_x_rho[:, 0]))
        Change_m2_x.append(modal_assurance_criterion(phi_x[:, 1], phi_x_rho[:, 1]))
        Change_m3_x.append(modal_assurance_criterion(phi_x[:, 2], phi_x_rho[:, 2]))
        Change_m1_y.append(modal_assurance_criterion(phi_y[:, 0], phi_y_rho[:, 0]))
        Change_m2_y.append(modal_assurance_criterion(phi_y[:, 1], phi_y_rho[:, 1]))
        Change_m3_y.append(modal_assurance_criterion(phi_y[:, 2], phi_y_rho[:, 2]))
    return Change_f1_x, Change_f2_x, Change_f3_x, Change_f1_y, Change_f2_y, Change_f3_y,
        Change_m1_x, Change_m2_x, Change_m3_x, Change_m1_y, Change_m2_y, Change_m3_y

----------------------------------------------------------------------------------------

import matplotlib.pyplot as plt

import matplotlib.pyplot as plt
import numpy as np

def plot_per_parameter_influence(type, parameter, direction, parameter_change, *data):
    plt.rcParams['font.family'] = 'Arial'

    plt.figure(figsize=(7, 5))
    colors = ['black', 'firebrick', 'steelblue']
    linestyles = ['-', '-', '-']
    mode = ['Mode 1x', 'Mode 1y', 'Mode 2y']
    for mod, color, linestyle, data_array in zip(mode, colors, linestyles, data):
        if type == 'frequency':
```

```
            if isinstance(data_array[0], (list, np.ndarray)):
                abs_data_array = [[abs(val) for val in sublist] for sublist in data_array]
            else:
                abs_data_array = [abs(val) for val in data_array]
            plt.plot(parameter_change, abs_data_array, label=mod, color=color, linestyle=linestyle, linewidth=1.2)
        else:
            plt.plot(parameter_change, data_array, label=mod, color=color, linestyle=linestyle, linewidth=1.2)
    plt.xlabel(f'Scalar applied on initial value parameter {parameter} (-)', fontsize=12)
    if type == 'frequency':
        plt.ylabel('Absolute change natural frequency $_{{fn}}$ (|%|)', fontsize=12)
        plt.ylim(-1.5, 100)
        plt.yticks(np.arange(0, 110, 10))
        plt.legend(loc='upper right')
    else:
        plt.ylabel('Change mode shape $_{{MAC}}$ (-)', fontsize=12)
        plt.ylim(0.70, 1.01)
        plt.yticks(np.arange(0.70, 1.05, 0.05))
        plt.legend(loc='lower right')
    plt.gca().spines['top'].set_linewidth(0.5)
    plt.gca().spines['top'].set_color('black')
    plt.gca().spines['right'].set_linewidth(0.5)
    plt.gca().spines['right'].set_color('black')
    plt.gca().spines['bottom'].set_linewidth(0.5)
    plt.gca().spines['bottom'].set_color('black')
    plt.gca().spines['left'].set_linewidth(0.5)
    plt.gca().spines['left'].set_color('black')
    plt.xscale('log')
    plt.gca().set_xticks([0.1, 1, 10])
    plt.gca().set_xticklabels(['0.1', '1', '10'])
    plt.grid(True, color='grey', linestyle=':')
    plt.show()

%load_ext autoreload
%autoreload 2
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam2D
from pyvibe.studies.influence_parameters import change_E, change_rho, change_kG, change_kr, change_I, change_kt, plot_per_parameter_influence
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

## Parameters

# Dimension Parameters:
L = 160.58 #[m]
Lt = np.array([5.735, 11.47, 26.640, 89.91, 26.825]) #[m]
B = 28 #[m]
D = 28 #[m]
A = B * D #[m2]
#I_x = np.array([619.21, 948.47, 2727.22, 2335.82, 905.52]) #[m4]
#I_y = np.array([1536.25, 959.01, 1245.46, 1202.07, 831.57]) #[m4]
I_yy = np.array([1444,1500,1393,1389,1293]) #[m4]
I_xx = np.array([1532,1548,2796,2324,760]) #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([1933, 495, 486, 440, 383]) #[kg/m3]
E = np.array([38.2e9, 38.2e9, 38.2e9, 38.2e9, 32.8e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v)) #[N/m^2]
k = 0.85 #[-]

# Springs:
kr_x = 2380e9 #[Nm/rad]
kr_y = 2625e9 #[Nm/rad]
kt_y = 19e9 #[N/m]
kt_x = 4e9 #[N/m]
```

```
#boudary conditions:
bc = ("rot+transl", "free")

# Scalar boundaries
amount_points = 300
left = 0.1
right = 10

## Procedure

timoshenko_x = []
for ix in range(len(H)):
    timoshenko_x.append(Timo(params={"E": E[ix], "I": I_xx[ix],"L": L,"rho": rho[ix], "A": A
        , "kG": k * G[ix], "kr": kr_x, "kt": kt_x}, height=H[ix]))
timoshenko_y = []
for ix in range(len(H)):
    timoshenko_y.append(Timo(params={"E": E[ix], "I": I_yy[ix],"L": L,"rho": rho[ix], "A": A
        , "kG": k * G[ix], "kr": kr_y, "kt": kt_y}, height=H[ix]))
model2D = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc = bc)
fn_x, fn_y, phi_x, phi_y, pts_x, pts_y = model2D.modeshapes()
change = np.logspace(np.log10(left), np.log10(right), amount_points)

Change_f1_x_kr, Change_f2_x_kr, Change_f3_x_kr, Change_f1_y_kr, Change_f2_y_kr,
    Change_f3_y_kr, Change_m1_x_kr, Change_m2_x_kr, Change_m3_x_kr, Change_m1_y_kr,
    Change_m2_y_kr, Change_m3_y_kr = change_kr(change, fn_x, fn_y, phi_x, phi_y, E, E, I_xx,
    I_yy, L, rho, A, k, G, G, kr_x, kr_y, kt_x, kt_y, H, bc)

Change_f1_x_kt, Change_f2_x_kt, Change_f3_x_kt, Change_f1_y_kt, Change_f2_y_kt,
    Change_f3_y_kt, Change_m1_x_kt, Change_m2_x_kt, Change_m3_x_kt, Change_m1_y_kt,
    Change_m2_y_kt, Change_m3_y_kt = change_kt(change, fn_x, fn_y, phi_x, phi_y, E, E, I_xx,
    I_yy, L, rho, A, k, G, G, kr_x, kr_y, kt_x, kt_y, H, bc)

Change_f1_x_I, Change_f2_x_I, Change_f3_x_I, Change_f1_y_I, Change_f2_y_I, Change_f3_y_I,
    Change_m1_x_I, Change_m2_x_I, Change_m3_x_I, Change_m1_y_I, Change_m2_y_I, Change_m3_y_I
    = change_I(change, fn_x, fn_y, phi_x, phi_y, E, E, I_xx, I_yy, L, rho, A, k, G, G, kr_x,
    kr_y, kt_x, kt_y, H, bc)

Change_f1_x_E, Change_f2_x_E, Change_f3_x_E, Change_f1_y_E, Change_f2_y_E, Change_f3_y_E,
    Change_m1_x_E, Change_m2_x_E, Change_m3_x_E, Change_m1_y_E, Change_m2_y_E, Change_m3_y_E
    = change_E(change, fn_x, fn_y, phi_x, phi_y, E, E, I_xx, I_yy, L, rho, A, k, G, G, kr_x,
    kr_y, kt_x, kt_y, H, bc)

Change_f1_x_kG, Change_f2_x_kG, Change_f3_x_kG, Change_f1_y_kG, Change_f2_y_kG,
    Change_f3_y_kG, Change_m1_x_kG, Change_m2_x_kG, Change_m3_x_kG, Change_m1_y_kG,
    Change_m2_y_kG, Change_m3_y_kG = change_kG(change, fn_x, fn_y, phi_x, phi_y, E, E, I_xx,
    I_yy, L, rho, A, k, G, G, kr_x, kr_y, kt_x, kt_y, H, bc)

Change_f1_x_rho, Change_f2_x_rho, Change_f3_x_rho, Change_f1_y_rho, Change_f2_y_rho,
    Change_f3_y_rho, Change_m1_x_rho, Change_m2_x_rho, Change_m3_x_rho, Change_m1_y_rho,
    Change_m2_y_rho, Change_m3_y_rho = change_rho(change, fn_x, fn_y, phi_x, phi_y, E, E,
    I_xx, I_yy, L, rho, A, k, G, G, kr_x, kr_y, kt_x, kt_y, H, bc)

## Results per parameter

plot_per_parameter_influence('frequency','kr','x', change, Change_f1_x_kr, Change_f2_x_kr)
plot_per_parameter_influence('mode','kr','x', change, Change_m1_x_kr, Change_m2_x_kr)
plot_per_parameter_influence('frequency','kr','y', change, Change_f1_y_kr, Change_f2_y_kr,
    Change_f3_y_kr)
plot_per_parameter_influence('mode','kr','y', change, Change_m1_y_kr, Change_m2_y_kr,
    Change_m3_y_kr)

plot_per_parameter_influence('frequency','kt', 'x', change, Change_f1_x_kt, Change_f2_x_kt)
plot_per_parameter_influence('mode','kt', 'x', change, Change_m1_x_kt, Change_m2_x_kt)
plot_per_parameter_influence('frequency','kt', 'y', change, Change_f1_y_kt, Change_f2_y_kt,
    Change_f3_y_kt)
plot_per_parameter_influence('mode','kt', 'y', change, Change_m1_y_kt, Change_m2_y_kt,
    Change_m3_y_kt)

plot_per_parameter_influence('frequency','I', 'x', change, Change_f1_x_I, Change_f2_x_I)
plot_per_parameter_influence('mode','I', 'x', change, Change_m1_x_I, Change_m2_x_I)
plot_per_parameter_influence('frequency','I', 'y', change, Change_f1_y_I, Change_f2_y_I,
```

```
    Change_f3_y_I)
plot_per_parameter_influence('mode','I', 'y', change, Change_m1_y_I, Change_m2_y_I,
    Change_m3_y_I)

plot_per_parameter_influence('frequency','E','x', change, Change_f1_x_E, Change_f2_x_E)
plot_per_parameter_influence('mode','E','x', change, Change_m1_x_E, Change_m2_x_E)
plot_per_parameter_influence('frequency','E','y', change, Change_f1_y_E, Change_f2_y_E,
    Change_f3_y_E)
plot_per_parameter_influence('mode','E','y', change, Change_m1_y_E, Change_m2_y_E,
    Change_m3_y_E)

plot_per_parameter_influence('frequency','kG','x', change, Change_f1_x_kG, Change_f2_x_kG)
plot_per_parameter_influence('mode','kG','x', change, Change_m1_x_kG, Change_m2_x_kG)
plot_per_parameter_influence('frequency','kG','y', change, Change_f1_y_kG, Change_f2_y_kG,
    Change_f3_y_kG)
plot_per_parameter_influence('mode','kG','y', change, Change_m1_y_kG, Change_m2_y_kG,
    Change_m3_y_kG)

plot_per_parameter_influence('frequency','rho', 'x', change, Change_f1_x_rho, Change_f2_x_rho
    )
plot_per_parameter_influence('mode','rho', 'x', change, Change_m1_x_rho, Change_m2_x_rho)
plot_per_parameter_influence('frequency','rho', 'y', change, Change_f1_y_rho, Change_f2_y_rho
    , Change_f3_y_rho)
plot_per_parameter_influence('mode','rho', 'y', change, Change_m1_y_rho, Change_m2_y_rho,
    Change_m3_y_rho)
```

Optimization Algorithm

```
%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam2D, UpdatingMethod, cost_func2D,
    update_model
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd

## Parameters

# Dimension Parameters:
L = 153.87 #[m]
Lt = np.array([3.15, 22.65, 46.8, 81.27]) #[m]
A = np.array([1320, 1320, 1320, 825]) #[m]
I_yy = np.array([4312, 4312, 4744, 1402]) #[m4]
I_xx = np.array([34350, 34350, 35105, 10685]) #[m4]
H = Lt/L #[-]
# Material properties:
rho = np.array([5070, 552, 426, 371]) #[kg/m3]
E = np.array([11e9, 11e9, 22e9, 33e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v))#[N/m^2]
k = 0.85 #[-]

# Springs:
kr_y = 6486e9 #[Nm/rad]
kr_x = 50371e9 #[Nm/rad]
kt_x = 9.22e9 #[N/m]
kt_y = 9.22e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")
scaler = {"kr":0.4, "kt":9.0, "I":1.5, "E":7.0, "rho":2.0} #The parameters chosen are from
    the conclusion of sensitivity study 2
zz_trgt = np.array([0, 69, 108.60, 144.60])  #The three locations where there are
    measurements taken of the New Orleans
z_trgt = zz_trgt/L

## Procedure
```

```python

timoshenko_x = []
for ix in range(len(H)):
    timoshenko_x.append(Timo(params={"E": E[ix], "I": I_xx[ix],"L": L,"rho": rho[ix], "A": A[
        ix] , "kG": k * G[ix], "kr": kr_x, "kt": kt_x}, height=H[ix]))
timoshenko_y = []
for ix in range(len(H)):
    timoshenko_y.append(Timo(params={"E": E[ix], "I": I_yy[ix],"L": L,"rho": rho[ix], "A": A[
        ix] , "kG": k * G[ix], "kr": kr_y, "kt": kt_y}, height=H[ix]))
model = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc = bc)

model.scale_params(scalers=scaler)
fn_x, fn_y, phi_x, phi_y, pts_x, pts_y = model.modeshapes(pts=z_trgt, n_modes_x = 1,
    n_modes_y = 2)
y_hat = {"z": z_trgt, "fn_x": deepcopy(model.fn_x), "fn_y": deepcopy(model.fn_y), "modes_x":
    deepcopy(model.phi_x), "modes_y": deepcopy(model.phi_y)}

parameters_to_update = {p:[1/10, 9.8] for p in list(scaler.keys())}
logscale = False

print(parameters_to_update)

def process_iteration(npts):
    results = update_model(
        model=model,
        cost_func=cost_func2D,
        parameters_to_update=parameters_to_update,
        y_hat=y_hat,
        npts=npts,
        sim_type=UpdatingMethod.CMC,
        verbose=False,
        logscale=logscale)

    parameters_columns = [cc for cc in results.columns if "_st" in cc or "_opt" in cc]
    plot_cols = [a + "_opt" for a in parameters_to_update.keys()]
    map_opt2name = {pcl: k for k, pcl in zip(parameters_to_update.keys(), plot_cols)}
    df_opt = results.copy()
    df_opt.dropna(inplace=True)

    if logscale:
        df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

    top_lowest_convergence = df_opt.sort_values(by='Convergence').head(1)
    return (npts,
            top_lowest_convergence["kr_opt"].iloc[0],
            top_lowest_convergence["kt_opt"].iloc[0],
            top_lowest_convergence["I_opt"].iloc[0],
            top_lowest_convergence["E_opt"].iloc[0],
            top_lowest_convergence["Convergence"].iloc[0],
            top_lowest_convergence["rho_opt"].iloc[0])

kr = []
kt = []
I = []
E = []
rho =[]
convergence = []
startpoints = []
npts_values = range(25, 525, 25)

for npts in npts_values:
    result = process_iteration(npts)
    startpoints.append(result[0])
    kr.append(result[1])
    kt.append(result[2])
    I.append(result[3])
    E.append(result[4])
    convergence.append(result[5])
    rho.append(result[6])

import os
```

```python
import pandas as pd
directory = r'C:\Users\ritfeldis\OneDrive␣-␣TNO\Documents\Master␣thesis␣Isa␣Ritfeld␣(4892534)\5.␣Python\5.3␣Results\Delfste␣Poort'
os.makedirs(directory, exist_ok=True)
file_names = [
    'CMC_3r_I.csv',
    'CMC_3r_E.csv',
    'CMC_3r_kr.csv',
    'CMC_3r_kt.csv',
    'CMC_3r_convergence.csv',
    'CMC_3r_startpoints.csv',
    'CMC_3r_rho.csv'
]
data_lists = [I, E, kr, kt, convergence, startpoints, rho]
parameter_names = ['I', 'E', 'kr', 'kt', 'convergence', 'startpoints', 'rho']

for file_name, data, parameter in zip(file_names, data_lists, parameter_names):
    df = pd.DataFrame({f'value␣of␣{parameter}': data})
    file_path = os.path.join(directory, file_name)
    df.to_csv(file_path, index=False)

directory = r'C:\Users\ritfeldis\OneDrive␣-␣TNO\Documents\Master␣thesis␣Isa␣Ritfeld␣(4892534)\5.␣Python\5.3␣Results\Delfste␣Poort'
file_names = [
    'combined_PSO_3_startpoints.csv',
    'combined_PSO_3_I.csv', #1
    'combined_PSO_3_E.csv', #2
    'combined_PSO_3_kr.csv', #3
    'combined_PSO_3_kt.csv', #4
    'combined_PSO_3_convergence.csv', #5
    'combined_DE_3_I.csv', #6
    'combined_DE_3_E.csv', #7
    'combined_DE_3_kr.csv', #8
    'combined_DE_3_kt.csv', #9
    'combined_DE_3_convergence.csv', #10
    'combined_GA_3_I.csv', #11
    'combined_GA_3_E.csv', #12
    'combined_GA_3_kr.csv', #13
    'combined_GA_3_kt.csv', #14
    'combined_GA_3_convergence.csv', #15
    'combined_CMC_3_I.csv', #16
    'combined_CMC_3_E.csv', #17
    'combined_CMC_3_kr.csv', #18
    'combined_CMC_3_kt.csv', #19
    'combined_CMC_3_convergence.csv', #20
    'combined_CMC_3r_I.csv', #21
    'combined_CMC_3r_E.csv', #22
    'combined_CMC_3r_kr.csv', #23
    'combined_CMC_3r_kt.csv', #24
    'combined_CMC_3r_convergence.csv', #25
]
dfs = []
for file_name in file_names:
    file_path = os.path.join(directory, file_name)
    df = pd.read_csv(file_path)
    dfs.append(df)

productPSO = dfs[1]['value␣of␣I'] * dfs[2]['value␣of␣E']
productGA = dfs[11]['value␣of␣I'] * dfs[12]['value␣of␣E']
productDE = dfs[6]['value␣of␣I'] * dfs[7]['value␣of␣E']
productCMC = dfs[16]['value␣of␣I'] * dfs[17]['value␣of␣E']
productCMCr = dfs[21]['value␣of␣I'] * dfs[22]['value␣of␣E']

plt.figure(figsize=(7, 5))

plt.plot(dfs[0], productPSO, color='lightsteelblue', label = 'PSO',marker='s', markersize=4.5,markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
plt.plot(dfs[0], productDE, color='black', label = 'DE',marker='v', markersize=6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-
```

```
')
plt.plot(dfs[0], productCMC, color='firebrick', label = 'CMC',marker='^', markersize=6, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
plt.plot(dfs[0], productCMCr, color='goldenrod', label='CMC␣with␣$\\rho$', marker='D', markersize=4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-', zorder =1)

# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')

# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

plt.ylim(7.5,13.5)
plt.yticks(np.arange(7.5, 13.6, 1.5), fontname='Arial')
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Set the y-axis height for the bottom spine
y_axis_height = 10.5
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.0)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use '' for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black')
# Set labels and legend
plt.ylabel('Scalar␣(-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount␣of␣candidate␣solutions␣(-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='upper␣right')

# Show the plot
plt.show()

## Results

import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(7, 5))

# Plot the data
plt.plot(dfs[0], dfs[1], color='lightsteelblue', label='PSO', marker='s', markersize=4.5, markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], dfs[11], color='steelblue', label='GA', marker='o', markersize=5, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', linestyle='-')
plt.plot(dfs[0], dfs[6], color='black', label='DE', marker='v', markersize=6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-')
plt.plot(dfs[0], dfs[16], color='firebrick', label='CMC', marker='^', markersize=6, linewidth
```

```
    =0.8, markeredgewidth=0.5, markeredgecolor='black', zorder=3)
plt.plot(dfs[0], dfs[21], color='goldenrod', label='CMC␣with␣$\\rho$', marker='D', markersize
    =4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-')

# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

plt.yticks(np.arange(0, 6.01, 1.5), fontname='Arial')
plt.ylim(0, 6.0)

# Set the y-axis height for the bottom spine
y_axis_height = 1.5
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.01)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use ''
    for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black'
    )
# Set labels and legend
plt.ylabel('Scalar␣(-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount␣of␣candidate␣solutions␣(-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='upper␣right')

# Show the plot
plt.show()

import matplotlib.pyplot as plt
import numpy as np

plt.figure(figsize=(7, 5))

# Plot the data
plt.plot(dfs[0], dfs[2], color='lightsteelblue', label='PSO', marker='s', markersize=4.5,
    markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], dfs[12], color='steelblue', label='GA', marker='o', markersize=5, linewidth
    =0.8, markeredgewidth=0.5, markeredgecolor='black', linestyle='-')
plt.plot(dfs[0], dfs[7], color='black', label='DE', marker='v', markersize=6, linewidth=0.8,
    markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-')
plt.plot(dfs[0], dfs[17], color='firebrick', label='CMC', marker='^', markersize=6, linewidth
    =0.8, markeredgewidth=0.5, markeredgecolor='black', zorder=3)
plt.plot(dfs[0], dfs[22], color='goldenrod', label='CMC␣with␣$\\rho$', marker='D', markersize
    =4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-')

# Customize the grid and spines
```

```
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

plt.yticks(np.arange(1, 11.01, 2), fontname='Arial')
plt.ylim(1, 11.0)

# Set the y-axis height for the bottom spine
y_axis_height = 7.0
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.01)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use ''
    for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
# Set labels and legend
plt.ylabel('Scalar␣(-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount␣of␣candidate␣solutions␣(-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='upper␣right')

# Show the plot
plt.show()

plt.figure(figsize=(7, 5))

plt.plot(dfs[0], dfs[3], color='lightsteelblue', label = 'PSO',marker='s', markersize=4.5,
    markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5,
    linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
plt.plot(dfs[0], dfs[8], color='black', label = 'DE',marker='v', markersize=6, linewidth=0.8,
     markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-')
plt.plot(dfs[0], dfs[18], color='firebrick', label = 'CMC',marker='^', markersize=6,
    linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
plt.plot(dfs[0], dfs[23], color='goldenrod', label='CMC␣with␣$\\rho$', marker='D', markersize
    =4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-',zorder =1)

# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')
```

```python

# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

plt.ylim(0.32,0.42)
plt.yticks(np.arange(0.32, 0.421, 0.02), fontname='Arial')

# Set the y-axis height for the bottom spine
y_axis_height = 0.4
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.01)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use '' for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black')
# Set labels and legend
plt.ylabel('Scalar (-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount of candidate solutions (-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='lower right')

# Show the plot
plt.show()

plt.figure(figsize=(7, 5))

plt.plot(dfs[0], dfs[4], color='lightsteelblue', label = 'PSO',marker='s', markersize=4.5, markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
plt.plot(dfs[0], dfs[9], color='black', label = 'DE',marker='v', markersize=6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-')
plt.plot(dfs[0], dfs[19], color='firebrick', label = 'CMC',marker='^', markersize=6, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
plt.plot(dfs[0], dfs[24], color='goldenrod', label='CMC with $\\rho$', marker='D', markersize=4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-',zorder =1)

# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')

# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

plt.ylim(7.5,9.9)
plt.yticks(np.arange(7.5, 10.01, 0.5), fontname='Arial')

# Set the y-axis height for the bottom spine
y_axis_height = 9.0
```

```
ax.spines['bottom'].set_position(('data', y_axis_height))
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.01)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use ''
    for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black'
    )
# Set labels and legend
plt.ylabel('Scalar (-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount of candidate solutions (-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='lower right')

# Show the plot
plt.show()
```

Amount of Measured Modal Properties

```
%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam2D, UpdatingMethod, cost_func2D,
    update_model
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd

## Parameters

# Dimension Parameters:
L = 153.87 #[m]
Lt = np.array([3.15, 22.65, 46.8, 81.27]) #[m]
A = np.array([1320, 1320, 1320, 825]) #[m]
I_yy = np.array([4312, 4312, 4744, 1402]) #[m4]
I_xx = np.array([34350, 34350, 35105, 10685]) #[m4]
H = Lt/L #[-]
# Material properties:
rho = np.array([5070, 552, 426, 371]) #[kg/m3]
E = np.array([11e9, 11e9, 22e9, 33e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v))#[N/m^2]
k = 0.85 #[-]

# Springs:
kr_y = 6486e9 #[Nm/rad]
kr_x = 50371e9 #[Nm/rad]
kt_x = 9.22e9 #[N/m]
kt_y = 9.22e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")
scaler = {"kr":0.4, "kt":9.0, "I":1.5, "E":7.0}
zz_trgt = np.array([0, 69, 108.60, 144.60])
z_trgt = zz_trgt/L

```

```
## Procedure

timoshenko_x = []
for ix in range(len(H)):
    timoshenko_x.append(Timo(params={"E": E[ix], "I": I_xx[ix],"L": L,"rho": rho[ix], "A": A[
        ix] , "kG": k * G[ix], "kr": kr_x, "kt": kt_x}, height=H[ix]))
timoshenko_y = []
for ix in range(len(H)):
    timoshenko_y.append(Timo(params={"E": E[ix], "I": I_yy[ix],"L": L,"rho": rho[ix], "A": A[
        ix] , "kG": k * G[ix], "kr": kr_y, "kt": kt_y}, height=H[ix]))
model = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc = bc)

model.scale_params(scalers=scaler)
fn_x, fn_y, phi_x, phi_y, pts_x, pts_y = model.modeshapes(pts=z_trgt, n_modes_x = 1,
    n_modes_y = 1)
y_hat = {"z": z_trgt, "fn_x": deepcopy(model.fn_x), "fn_y": deepcopy(model.fn_y), "modes_x":
    deepcopy(model.phi_x), "modes_y": deepcopy(model.phi_y)}

parameters_to_update = {p:[1/10, 9.8] for p in list(scaler.keys())}
logscale = False

print(parameters_to_update)

def process_iteration(npts):
    results = update_model(
        model=model,
        cost_func=cost_func2D,
        parameters_to_update=parameters_to_update,
        y_hat=y_hat,
        npts=npts,
        sim_type=UpdatingMethod.CMC,
        verbose=False,
        logscale=logscale)

    parameters_columns = [cc for cc in results.columns if "_st" in cc or "_opt" in cc]
    plot_cols = [a + "_opt" for a in parameters_to_update.keys()]
    map_opt2name = {pcl: k for k, pcl in zip(parameters_to_update.keys(), plot_cols)}
    df_opt = results.copy()
    df_opt.dropna(inplace=True)

    if logscale:
        df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

    top_lowest_convergence = df_opt.sort_values(by='Convergence').head(1)
    return (npts,
            top_lowest_convergence["kr_opt"].iloc[0],
            top_lowest_convergence["kt_opt"].iloc[0],
            top_lowest_convergence["I_opt"].iloc[0],
            top_lowest_convergence["E_opt"].iloc[0],
            top_lowest_convergence["Convergence"].iloc[0])

kr = []
kt = []
I = []
E = []
convergence = []
startpoints = []
npts_values = range(25, 525, 25)

for npts in npts_values:
    result = process_iteration(npts)
    startpoints.append(result[0])
    kr.append(result[1])
    kt.append(result[2])
    I.append(result[3])
    E.append(result[4])
    convergence.append(result[5])

import os
import pandas as pd
directory = r'C:\Users\ritfeldis\OneDrive - TNO\Documents\Master thesis Isa Ritfeld (4892534)
```

```
\5.␣Python\5.3␣Results\Delfste␣Poort'
os.makedirs(directory, exist_ok=True)
file_names = [
    'CMC_1_I.csv',
    'CMC_1_E.csv',
    'CMC_1_kr.csv',
    'CMC_1_kt.csv',
    'CMC_1_convergence.csv',
    'CMC_1_startpoints.csv'
]
data_lists = [I, E, kr, kt, convergence, startpoints]
parameter_names = ['I', 'E', 'kr', 'kt', 'convergence', 'startpoints']

for file_name, data, parameter in zip(file_names, data_lists, parameter_names):
    df = pd.DataFrame({f'value␣of␣{parameter}': data})
    file_path = os.path.join(directory, file_name)
    df.to_csv(file_path, index=False)

%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam2D, UpdatingMethod, cost_func2D, update_model
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import os
%matplotlib inline

## Parameters

directory = r'C:\Users\ritfeldis\OneDrive␣-␣TNO\Documents\Master␣thesis␣Isa␣Ritfeld␣(4892534)\5.␣Python\5.3␣Results\Delfste␣Poort'
file_names = [
    'combined_CMC_1_startpoints.csv',#0
    'combined_CMC_1_I.csv', #1
    'combined_CMC_1_E.csv', #2
    'combined_CMC_1_kr.csv', #3
    'combined_CMC_1_kt.csv', #4
    'combined_CMC_1_convergence.csv', #5
    'combined_CMC_3_I.csv', #6
    'combined_CMC_3_E.csv', #7
    'combined_CMC_3_kr.csv', #8
    'combined_CMC_3_kt.csv', #9
    'combined_CMC_3_convergence.csv', #10
]
dfs = []
for file_name in file_names:
    file_path = os.path.join(directory, file_name)
    df = pd.read_csv(file_path)
    dfs.append(df)

productCMC1 = dfs[1]['value␣of␣I'] * dfs[2]['value␣of␣E']
productCMC3 = dfs[6]['value␣of␣I'] * dfs[7]['value␣of␣E']

plt.figure(figsize=(7, 5))

plt.plot(dfs[0], productCMC1, color='lightsteelblue', label = '1␣in␣x␣and␣1␣in␣y',marker='s', markersize=4.5,markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
#plt.plot(dfs[0], productCMC2, color='black', label = '2 in x and 2 in y',marker='v', markersize=6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-')
plt.plot(dfs[0], productCMC3, color='firebrick', label = '1␣in␣x␣and␣2␣in␣y',marker='^', markersize=6, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
#plt.plot(dfs[0], productCMCr, color='goldenrod', label='CMC with $\\rho$', marker='D', markersize=4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-', zorder =1)

```

```
# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')

# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

#plt.ylim(9.80,10.6)
plt.yticks(np.arange(4.5, 19.6, 3), fontname='Arial')
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Set the y-axis height for the bottom spine
y_axis_height = 10.5
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.0)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use '' for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black')
# Set labels and legend
plt.ylabel('Scalar (-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount of candidate solutions (-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='lower right')

# Show the plot
plt.show()

plt.figure(figsize=(7, 5))

plt.plot(dfs[0], dfs[1], color='lightsteelblue', label = '1 in x and 1 in y',marker='s', markersize=4.5,markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
#plt.plot(dfs[0], dfs[6], color='black', label = '2 in x and 2 in y',marker='v', markersize=6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-')
plt.plot(dfs[0], dfs[6], color='firebrick', label = '1 in x and 2 in y',marker='^', markersize=6, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
#plt.plot(dfs[0], productCMCr, color='goldenrod', label='CMC with $\\rho$', marker='D', markersize=4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-', zorder =1)
# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
```

```
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

ax.set_yticks(np.arange(0.5, 2.6, 0.5))
ax.set_ylim(0.5, 2.5)

# Set the y-axis height for the bottom spine
y_axis_height = 1.5
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.0)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use ''
    for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black'
    )
# Set labels and legend
plt.ylabel('Scalar (-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount of candidate solutions (-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='upper right')

# Show the plot
plt.show()

plt.figure(figsize=(7, 5))

plt.plot(dfs[0], dfs[2], color='lightsteelblue', label = '1 in x and 1 in y',marker='s',
    markersize=4.5,markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5,
    linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
#plt.plot(dfs[0], dfs[7], color='black', label = '2 in x and 2 in y',marker='v', markersize
    =6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white',
     linestyle='-')
plt.plot(dfs[0], dfs[7], color='firebrick', label = '1 in x and 2 in y',marker='^',
    markersize=6, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
#plt.plot(dfs[0], productCMCr, color='goldenrod', label='CMC with $\\rho$', marker='D',
    markersize=4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-',
    zorder =1)

# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')

# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

```

```
ax.set_yticks(np.arange(3, 11.1, 2))
ax.set_ylim(3, 11)

# Set the y-axis height for the bottom spine
y_axis_height = 7.0
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.0)  # Adjust x-axis label position
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use '' for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black')
# Set labels and legend
plt.ylabel('Scalar␣(-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount␣of␣candidate␣solutions␣(-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='upper␣right')

# Show the plot
plt.show()

plt.figure(figsize=(7, 5))
plt.plot(dfs[0], dfs[3], color='lightsteelblue', label = '1␣in␣x␣and␣1␣in␣y',marker='s', markersize=4.5,markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
#plt.plot(dfs[0], dfs[13], color='black', label = '2 in x and 2 in y',marker='v', markersize=6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white', linestyle='-')
plt.plot(dfs[0], dfs[8], color='firebrick', label = '1␣in␣x␣and␣2␣in␣y',marker='^', markersize=6, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
#plt.plot(dfs[0], productCMCr, color='goldenrod', label='CMC with $\\rho$', marker='D', markersize=4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-', zorder =1)

# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')

# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

#ax.set_yticks(np.arange(0, 14.1, 2))
#ax.set_ylim(0.0, 14)
ax.set_yticks(np.arange(0.36, 0.481, 0.02))
#ax.set_ylim(0.3990, 0.403)
# Set the y-axis height for the bottom spine
y_axis_height = 0.4
ax.spines['bottom'].set_position(('data', y_axis_height))
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
```

```
# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.0)  # Adjust x-axis label position

# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use ''
    for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black'
    )
# Set labels and legend
plt.ylabel('Scalar␣(-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount␣of␣candidate␣solutions␣(-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='upper␣right')

# Show the plot
plt.show()

plt.figure(figsize=(7, 5))

plt.plot(dfs[0], dfs[4], color='lightsteelblue', label = '1␣in␣x␣and␣1␣in␣y',marker='s',
    markersize=4.5,markeredgewidth=0.5, markeredgecolor='black', linewidth=1, linestyle='-')
#plt.plot(dfs[0], productGA, color='steelblue', label = 'GA',marker='o', markersize=5,
    linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black',  linestyle='-')
#plt.plot(dfs[0], dfs[14], color='black', label = '2 in x and 2 in y',marker='v', markersize
    =6, linewidth=0.8, markeredgewidth=0.5, markeredgecolor='black', markerfacecolor='white',
     linestyle='-')
plt.plot(dfs[0], dfs[9], color='firebrick', label = '1␣in␣x␣and␣2␣in␣y',marker='^',
    markersize=6, linewidth=0.8,markeredgewidth=0.5, markeredgecolor='black', zorder=3)
#plt.plot(dfs[0], productCMCr, color='goldenrod', label='CMC with $\\rho$', marker='D',
    markersize=4, markeredgewidth=0.5, markeredgecolor='black', linewidth=0.8, linestyle='-',
    zorder =1)

# Customize the grid and spines
plt.grid(True, which='both', linestyle=':', linewidth='0.5', color='white')
ax = plt.gca()
ax.spines['top'].set_linewidth(0.5)
ax.spines['top'].set_color('white')
ax.spines['right'].set_linewidth(0.5)
ax.spines['right'].set_color('white')
ax.spines['bottom'].set_linewidth(0.5)
ax.spines['bottom'].set_color('black')
ax.spines['left'].set_linewidth(0.5)
ax.spines['left'].set_color('black')

# Set tick positions and limits
plt.xticks(np.arange(0, 501, 100), fontname='Arial')

#ax.set_yticks(np.arange(0, 14.1, 2))
#ax.set_ylim(0.0, 14)
ax.set_yticks(np.arange(7, 11.1, 1))
ax.set_ylim(7, 11)
# Set the y-axis height for the bottom spine
y_axis_height = 9.0
ax.spines['bottom'].set_position(('data', y_axis_height))

# Manually adjust x-axis label position
ax.xaxis.set_label_coords(-0.5, 0.0)  # Adjust x-axis label position
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('%.2f'))
# Ensure a tick mark at the end of x-axis without a corresponding value
xticks = np.arange(100, 501, 100)
plt.xlim(15, 525)
```

```
xticks = np.append(xticks, 525)  # Add the endpoint where you want the tick without a label
ax.set_xticks(xticks)
ax.set_xticklabels([str(int(x)) if x != 525 else '' for x in xticks])  # Set labels; use ''
    for the endpoint

# Customize the ticks
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='out', length=2, width=1, colors='black')
plt.gca().yaxis.set_ticks_position('left')
plt.gca().xaxis.set_ticks_position('bottom')# Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=2, width=1, colors='black')
plt.tick_params(axis='x', which='major', direction='inout', length=4, width=1, colors='black'
    )
# Set labels and legend
plt.ylabel('Scalar (-)', fontsize=10, fontname='Arial')
plt.xlabel('Amount of candidate solutions (-)', fontsize=10, fontname='Arial')
plt.legend(prop={'family': 'Arial'}, loc='upper right')

# Show the plot
plt.show()
```

Measurement Uncertainties

```
import time
%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam1D, TimoPieceWiseBeam2D,
    UpdatingMethod, cost_func1D, cost_func2D,update_model
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# Dimension Parameters:
L = 153.87 #[m]
Lt = np.array([3.15, 22.65, 46.8, 81.27]) #[m]
A = np.array([1320, 1320, 1320, 825]) #[m]
I_yy = np.array([4312, 4312, 4744, 1402]) #[m4]
I_xx = np.array([34350, 34350, 35105, 10685]) #[m4]
H = Lt/L #[-]
# Material properties:
rho = np.array([5070, 552, 426, 371]) #[kg/m3]
E = np.array([11e9, 11e9, 22e9, 33e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v))#[N/m^2]
k = 0.85 #[-]

# Springs:
kr_y = 6486e9 #[Nm/rad]
kr_x = 50371e9 #[Nm/rad]
kt_x = 9.22e9 #[N/m]
kt_y = 9.22e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")
scaler = {"kr":0.4, "kt":9.0, "I":1.5, "E":7.0}
zz_trgt = np.array([0, 69, 108.60, 144.60])
z_trgt = zz_trgt/L

#Startpoints
startpoints = 200

# Number of iterations
num_iterations = 10

timoshenko_x = []
for ix in range(len(H)):
    timoshenko_x.append(Timo(params={"E": E[ix], "I": I_xx[ix],"L": L,"rho": rho[ix], "A": A[
        ix] , "kG": k * G[ix], "kr": kr_x, "kt": kt_x}, height=H[ix]))
timoshenko_y = []
```

```
for ix in range(len(H)):
    timoshenko_y.append(Timo(params={"E": E[ix], "I": I_yy[ix],"L": L,"rho": rho[ix], "A": A[
        ix] , "kG": k * G[ix], "kr": kr_y, "kt": kt_y}, height=H[ix]))
model = TimoPieceWiseBeam2D(beams_x=timoshenko_x, beams_y=timoshenko_y, bc = bc)

import numpy as np

def apply_random_errors(input_data, max_error=0.2, seed=None):
    if seed is not None:
        np.random.seed(seed)

    if input_data.ndim == 1:
        data_with_errors = np.copy(input_data)
        scaling_factors = np.random.uniform(1 - max_error, 1 + max_error, input_data.shape)
        data_with_errors *= scaling_factors

    elif input_data.ndim == 2:
        data_with_errors = np.copy(input_data)
        scaling_factors = np.random.uniform(1 - max_error, 1 + max_error, input_data.shape)
        data_with_errors *= scaling_factors

    else:
        raise ValueError("Input data must be either a 1D or 2D array.")

    return data_with_errors

model.scale_params(scalers=scaler)
fn_x, fn_y, phi_x, phi_y, pts_x, pts_y = model.modeshapes(pts=z_trgt, n_modes_x = 1,
    n_modes_y=2)
parameters_to_update = {p:[1/10, 9.8] for p in list(scaler.keys())}
logscale = False

def process_iteration(npts):
    results = update_model(
        model=model,
        cost_func=cost_func2D,
        parameters_to_update=parameters_to_update,
        y_hat=y_hat,
        npts=npts,
        sim_type=UpdatingMethod.CMC,
        verbose=False,
        logscale=logscale)

    parameters_columns = [cc for cc in results.columns if "_st" in cc or "_opt" in cc]
    plot_cols = [a + "_opt" for a in parameters_to_update.keys()]
    map_opt2name = {pcl: k for k, pcl in zip(parameters_to_update.keys(), plot_cols)}
    df_opt = results.dropna().copy()
    df_opt.dropna(inplace=True)

    if logscale:
        df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

    top_lowest_convergence = df_opt.sort_values(by='Convergence').head(1)
    return (npts,
            top_lowest_convergence["kr_opt"].iloc[0],
            top_lowest_convergence["kt_opt"].iloc[0],
            top_lowest_convergence["I_opt"].iloc[0],
            top_lowest_convergence["E_opt"].iloc[0],
            top_lowest_convergence["Convergence"].iloc[0])

kr = []
kt = []
I = []
E = []
convergence = []

for i in range(num_iterations):
    seed_fn_xt = i * 4
    seed_phi_xt = i * 4 + 1
    seed_fn_yt = i * 4 + 2
    seed_phi_yt = i * 4 + 3
```

```

    fn_xt = apply_random_errors(deepcopy(fn_x), max_error=0.1, seed=seed_fn_xt)
    phi_xt = apply_random_errors(deepcopy(phi_x), max_error=0.2, seed=seed_phi_xt)
    fn_yt = apply_random_errors(deepcopy(fn_y), max_error=0.1, seed=seed_fn_yt)
    phi_yt = apply_random_errors(deepcopy(phi_y), max_error=0.2, seed=seed_phi_yt)

    y_hat = {"z": z_trgt, "fn_x": deepcopy(fn_xt), "fn_y": deepcopy(fn_yt), "modes_x": deepcopy(phi_xt), "modes_y": deepcopy(phi_yt)}

    result = process_iteration(startpoints)

    kr.append(result[1])
    kt.append(result[2])
    I.append(result[3])
    E.append(result[4])
    convergence.append(result[5])

import os
import pandas as pd
directory = r'C:\Users\ritfeldis\OneDrive␣-␣TNO\Documents\Master␣thesis␣Isa␣Ritfeld␣(4892534)\5.␣Python\5.3␣Results\Delfste␣Poort'
os.makedirs(directory, exist_ok=True)
file_names = [
    'Measure_I.csv',
    'Measure_E.csv',
    'Measure_kr.csv',
    'Measure_kt.csv',
    'Measure_convergence.csv',
]
data_lists = [I, E, kr, kt, convergence]
parameter_names = ['I', 'E', 'kr', 'kt', 'convergence']

for file_name, data, parameter in zip(file_names, data_lists, parameter_names):
    df = pd.DataFrame({f'value␣of␣{parameter}': data})
    file_path = os.path.join(directory, file_name)
    df.to_csv(file_path, index=False)

directory = r'C:\Users\ritfeldis\OneDrive␣-␣TNO\Documents\Master␣thesis␣Isa␣Ritfeld␣(4892534)\5.␣Python\5.3␣Results\Delfste␣Poort'
file_names = [
    'Measure_I.csv',
    'Measure_E.csv',
    'Measure_kr.csv',
    'Measure_kt.csv',
    'Measure_convergence.csv',
]
dfs = []
for file_name in file_names:
    file_path = os.path.join(directory, file_name)
    df = pd.read_csv(file_path)
    dfs.append(df)

import os
import pandas as pd
import matplotlib.pyplot as plt

# Define the directory and file names
directory = r'C:\Users\ritfeldis\OneDrive␣-␣TNO\Documents\Master␣thesis␣Isa␣Ritfeld␣(4892534)\5.␣Python\5.3␣Results\Delfste␣Poort'
file_names = [
    'Measure_I.csv',
    'Measure_E.csv',
    'Measure_kr.csv',
    'Measure_kt.csv',
    'Measure_convergence.csv',
]
column_names = ['I', 'E', 'kr', 'kt', 'convergence']

# Load the data into a list of DataFrames
dfs = []
for file_name, col_name in zip(file_names, column_names):
```

```
    file_path = os.path.join(directory, file_name)
    df = pd.read_csv(file_path)
    df.columns = [col_name]
    dfs.append(df)

# Merge all DataFrames on the index
merged_df = pd.concat(dfs, axis=1)

# Add 'EI_opt' column which is the product of 'E' and 'I'
merged_df['EI'] = merged_df['E'] * merged_df['I']
print(merged_df['EI'])
# Define the additional solution
additional_solution = {
    'kr': 0.4,
    'kt': 9.0,
    'E': 7.0,
    'I': 1.5,
    'EI': 7.0 * 1.5
}

# Define columns to plot
columns = ['kr', 'kt', 'E', 'I', 'EI']

# Define colors for each column
colors = {
    'kr': 'steelblue',
    'kt': 'lightsteelblue',
    'I': 'firebrick',
    'E': 'goldenrod',
    'EI': 'white'
}

# Define marker styles for each column
markers = {
    'kr': 'o',      # circle
    'kt': 's',      # square
    'I': '^',       # triangle up
    'E': 'D',       # diamond
    'EI': 'v'  # plus (filled)
}

# Define marker sizes for each column
marker_sizes = {
    'kr': 50,       # Adjust marker size as needed
    'kt': 43,
    'I': 60,
    'E': 35,
    'EI': 60
}

# Define the additional solution color
additional_color = 'red'

# Set up the plot
plt.figure(figsize=(9, 6))

# Create scatter plot for each column with thin black edges
for col in columns:
    plt.scatter(
        [col] * len(merged_df),
        merged_df[col],
        label=col,
        color=colors[col],
        marker=markers[col],
        edgecolor='black',
        linewidth=0.5,
        s=marker_sizes[col],
        zorder=2
    )

# Connect the dots for each row
```

```
for i in range(len(merged_df)):
    plt.plot(columns, merged_df.iloc[i][columns], color='dimgrey', linestyle='-', linewidth
        =0.5, zorder=1)

# Plot the additional solution with corresponding markers and additional color
for col in columns:
    plt.scatter(
        [col],
        [additional_solution[col]],
        color=additional_color,
        marker=markers[col],
        edgecolor='black',
        linewidth=0.5,
        s=marker_sizes[col],
        zorder=3
    )
plt.plot(columns, list(additional_solution.values()), color=additional_color, linestyle='-',
    linewidth=0.5, zorder=3)

# Add labels and title
plt.xlabel('Optimized␣updating␣parameters', fontsize=12)
plt.ylabel('Scalar␣(-)', fontsize=12)
plt.xticks(rotation=45)

# Customize grid
plt.grid(True, which='major', axis='x', linestyle='-', linewidth=0.5, color='black')  # Major
     vertical grid lines

# Hide major horizontal grid lines
plt.grid(True, which='major', axis='y', linestyle='-', linewidth=0.5, color='white')

# Add ticks where vertical grid lines intersect horizontal lines
for col in columns:
    y_vals = [0.1, 1, 10, 20]
    for y_val in y_vals:
        plt.scatter(col, y_val, color='black', marker='_', linewidths=1.7, s=40, zorder=0)

for col in columns:
    y_vals = [0.2,0.3,0.4, 0.5,0.6,0.7,0.8,0.9, 2,3,4,5,6,7,8,9]
    for y_val in y_vals:
        plt.scatter(col, y_val, color='black', marker='_', linewidths=0.5, s=2, zorder=0)

# Adjust spines
plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('white')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('white')
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('white')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.gca().yaxis.set_ticks_position('left') # Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=5, width=1, colors='black')
     # Customize tick parameters

# Set y-scale and limits
plt.yscale('log')
plt.ylim(0.1, 20)
plt.gca().set_yticks([0.1, 1, 10, 20])
plt.gca().set_yticklabels(['0.1', '1', '10', '20'])

# Show legend for main columns only
plt.legend(columns, bbox_to_anchor=(0.05, 1), loc='upper␣left', prop={'family': 'Arial', '
    size': 10})

# Display the plot
plt.tight_layout()
plt.show()
```

Application

```
%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam1D, TimoPieceWiseBeam2D,
    UpdatingMethod, cost_func1D, cost_func2D,update_model
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
import math
import pandas as pd
%matplotlib inline

# Parameters

# Dimension Parameters:
L = 160.58 #[m]
Lt = np.array([5.735, 11.47, 26.640, 89.91, 26.825]) #[m]
B = 28 #[m]
D = 28 #[m]
A = B * D #[m2]
#I_x = np.array([619.21, 948.47, 2727.22, 2335.82, 905.52]) #[m4]
#I_y = np.array([1536.25, 959.01, 1245.46, 1202.07, 831.57]) #[m4]
I_yy = np.array([1444,1500,1393,1389,1293]) #[m4]
I_xx = np.array([1532,1548,2796,2324,760]) #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([1933, 495, 486, 440, 383]) #[kg/m3]
E = np.array([38.2e9, 38.2e9, 38.2e9, 38.2e9, 32.8e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v)) #[N/m^2]
k = 0.85 #[-]

# Springs:
kr_x = 2380e9 #[Nm/rad]
kr_y = 2625e9 #[Nm/rad]
kt_y = 19e9 #[N/m]
kt_x = 4e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")

scaler = {"kr":0.4, "kt":9.0, "I":1.5, "E":7.0} #The parameters chosen are from the
    conclusion of sensitivity study 2
zz_trgt = np.array([57.165, 120.435, 153.735])  #The three locations where there are
    measurements taken of the New Orleans
z_trgt = zz_trgt/L

#Startpoints
startpoints = 400

fnx = np.array([0.282, 1.527])
phixx = [
    [-0.30, 0.89],
    [-0.71, -0.23],
    [-1, -1]
]
phix = np.array(phixx)

# Procedure

def adjust_modes(phi1, phi2):
    rmse_plus = np.sqrt(np.mean((phi1 - phi2) ** 2))
    rmse_minus = np.sqrt(np.mean((phi1 + phi2) ** 2))
    flip = np.ones_like(phi2)
    flip[rmse_plus > rmse_minus] = -1
    phi_flip = phi1 * flip
```

```
    return phi_flip

def modal_assurance_criterion(phi1, phi2):
    phi2 = adjust_modes(phi2, phi1)
    mac = np.abs(np.conj(phi1).dot(phi2))**2 / \
              (np.conj(phi1).dot(phi1) * np.conj(phi2).dot(phi2))
    return mac

timoshenko_x = []
for ix in range(len(H)):
    timoshenko_x.append(Timo(params={"E": E[ix], "I": I_xx[ix],"L": L,"rho": rho[ix], "A": A
        , "kG": k * G[ix], "kr": kr_x, "kt": kt_x}, height=H[ix]))
model_x = TimoPieceWiseBeam1D(beams=timoshenko_x, bc = bc)

y_hat_x = {"z": z_trgt, "fn": deepcopy(fnx), "modes": deepcopy(phix)}
parameters_to_update_x = {p:[1/10, 10] for p in list(scaler.keys())} #maybe change this?
print(parameters_to_update_x)
logscale = False

results_x = update_model(
    model=model_x,
    cost_func=cost_func1D,
    parameters_to_update=parameters_to_update_x,
    y_hat=y_hat_x,
    npts=startpoints,
    sim_type=UpdatingMethod.CMC,
    verbose=False,
    logscale=logscale
)
parameters_columns = [cc for cc in results_x.columns if "_st" in cc or "_opt" in cc]
plot_cols = [a + "_opt" for a in parameters_to_update_x.keys()]
map_opt2name = {pcl: k for k, pcl in zip(parameters_to_update_x.keys(), plot_cols)}
df_opt = results_x.copy()
df_opt.dropna(inplace=True)

if logscale:
    df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

# Results

t_x = results_x.sort_values(by='Convergence').head(startpoints)

quantile_threshold = t_x['Convergence'].quantile(0.125)
test_x = t_x[t_x['Convergence'] <= quantile_threshold]
print(test_x)

# Find the minimum and maximum values in each _opt column
min_values = test_x[[col for col in test_x.columns if col.endswith('_opt')]].min()
max_values = test_x[[col for col in test_x.columns if col.endswith('_opt')]].max()

# Update the boundaries
new_boundaries = {
    'kr': [min_values['kr_opt'], max_values['kr_opt']],
    'kt': [min_values['kt_opt'], max_values['kt_opt']],
    'I': [min_values['I_opt'], max_values['I_opt']],
    'E': [min_values['E_opt'], max_values['E_opt']]
}
print("New␣Boundaries:")
print(new_boundaries)

results_repeated2_x = update_model(
    model=model_x,
    cost_func=cost_func1D,
    parameters_to_update=new_boundaries,
    y_hat=y_hat_x,
    npts=startpoints,
    sim_type=UpdatingMethod.CMC,
    verbose=False,
    logscale=logscale
)
parameters_columns = [cc for cc in results_repeated2_x.columns if "_st" in cc or "_opt" in cc
```

```
    ]
plot_cols = [a + "_opt" for a in new_boundaries.keys()]
map_opt2name = {pcl: k for k, pcl in zip(new_boundaries.keys(), plot_cols)}
df_opt = results_repeated2_x.copy()
df_opt.dropna(inplace=True)

if logscale:
    df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

t2_x = results_repeated2_x.sort_values(by='Convergence').head(startpoints)
print(t2_x)

quantile_threshold = t2_x['Convergence'].quantile(0.125)
test2_x = t2_x[t2_x['Convergence'] <= quantile_threshold]
print(test2_x)

results_output_x = []
for ii in range(len(test2_x)):
    scaler_x = {
        "kr": test2_x["kr_opt"].iloc[ii],
        "kt": test2_x["kt_opt"].iloc[ii],
        "I": test2_x["I_opt"].iloc[ii],
        "E": test2_x["E_opt"].iloc[ii]
    }
    model_x.scale_parms(scalers=scaler_x)
    fn_x, phi_x, pts_x = model_x.modeshapes(pts=z_trgt, n_modes=2)

    mac_0 = modal_assurance_criterion(phi_x[:, 0], phix[:, 0])
    mac_1 = modal_assurance_criterion(phi_x[:, 1], phix[:, 1])

    results_output_x.append({
        "index": ii,
        "fn_x_1": fn_x[0],
        "fn_x_2": fn_x[1],
        "mac_1": mac_0,
        "mac_2": mac_1
    })

results_df = pd.DataFrame(results_output_x)
stats_results = {}
z_value = 1.645

for col in ['fn_x_1', 'fn_x_2', 'mac_1', 'mac_2']:
    mean_val = results_df[col].mean()
    median_val = results_df[col].median()
    max_val = results_df[col].max()
    min_val = results_df[col].min()

    stats_results[col] = {
        "Median": median_val,
        "Mean": mean_val,
        "lower_bound": min_val,
        "upper_bound": max_val
    }
summary_df = pd.DataFrame(columns=["Measured fnx", "Median", "Mean", "Lower Bound", "Upper 
    Bound"])
for param, values in stats_results.items():
    if "fn_x_1" in param:
        measured_fnx = fnx[0]
    elif "fn_x_2" in param:
        measured_fnx = fnx[1]
    else:
        measured_fnx = np.nan

    summary_df.loc[param] = [measured_fnx, values["Median"], values["Mean"], values["
        lower_bound"],
                             values["upper_bound"]]
print(summary_df)

def calculate_summary_stats(df, columns, direction):
    print(f"{direction}-direction")
```

```

    kr_summary_stats = []
    kt_summary_stats = []

    for column in columns:
        if "kr_opt" in column:
            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_boundkr = df[column].min()
            upper_boundkr = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            kr_summary_stats.append(pd.DataFrame({
                'Design': 1,
                'Median': median_val,
                'Lower Bound': lower_boundkr,
                'Upper Bound': upper_boundkr,
                'COV (%)': cv
            }, index=["kr_opt"]))

        elif "kt_opt" in column:
            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_boundkt = df[column].min()
            upper_boundkt = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            kt_summary_stats.append(pd.DataFrame({
                'Design': 1,
                'Median': median_val,
                'Lower Bound': lower_boundkt,
                'Upper Bound': upper_boundkt,
                'COV (%)': cv
            }, index=["kt_opt"]))

    kr_summary_stats_df = pd.concat(kr_summary_stats, axis=0)
    kt_summary_stats_df = pd.concat(kt_summary_stats, axis=0)

    print("Summary statistics for boundary conditions:")
    print(kr_summary_stats_df)
    print(kt_summary_stats_df.to_string(header=None))

    for ii in range(len(E)):
        print("")
        print(f"Summary statistics for discrete element {ii}")
        summary_stats_list = []

        for column in columns:
            if "kr_opt" in column or "kt_opt" in column:
                continue

            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_bound = df[column].min()
            upper_bound = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)
```

```
            if 'y' in direction:
                if "E_opt" in column:
                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val,
                        'Lower Bound': lower_bound ,
                        'Upper Bound': upper_bound,
                        'COV (%)': cv
                    }, index=["E_opt"])
                    summary_stats_list.append(summary_stats)
                elif "I_opt" in column:
                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val,
                        'Lower Bound': lower_bound ,
                        'Upper Bound': upper_bound,
                        'COV (%)': cv
                    }, index=["I_opt"])
                    summary_stats_list.append(summary_stats)
                    new_column = df["E_opt"] * df["I_opt"]
                    mean_val_new = new_column.mean()
                    median_val_new = new_column.median()
                    std_dev_new = new_column.std()

                    lower_bound_new = new_column.min()
                    upper_bound_new = new_column.max()

                    cv_new = round((std_dev_new / mean_val_new) * 100, 2)

                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val_new,
                        'Lower Bound': lower_bound_new ,
                        'Upper Bound': upper_bound_new,
                        'COV (%)': cv_new
                    }, index=["EI_opt"])
                    summary_stats_list.append(summary_stats)

        summary_stats_df = pd.concat(summary_stats_list, axis=0)
        print(summary_stats_df)

parameters_columns_y = [cc for cc in test2_x.columns if "_opt" in cc]
summary_stats_y = calculate_summary_stats(test2_x, parameters_columns_y, "y")

import matplotlib.pyplot as plt

# Assuming test2_x is already defined and has the columns 'kr_opt', 'kt_opt', 'I_opt', 'E_opt', and 'EI_opt'

# Add a new column 'EI_opt' which is the product of 'E_opt' and 'I_opt'
test2_x['EI_opt'] = test2_x['E_opt'] * test2_x['I_opt']

# Columns to plot
columns = ['kr_opt', 'kt_opt', 'E_opt', 'I_opt', 'EI_opt']

# Define colors for each column
colors = {
    'kr_opt': 'steelblue',
    'kt_opt': 'lightsteelblue',
    'I_opt': 'firebrick',
    'E_opt': 'goldenrod',
    'EI_opt': 'white'
}

# Define marker styles for each column
markers = {
    'kr_opt': 'o',      # circle
    'kt_opt': 's',      # square
    'I_opt': '^',       # triangle up
    'E_opt': 'D',       # diamond
```

```
    'EI_opt': 'v'      # plus (filled)
}

# Define marker sizes for each column
marker_sizes = {
    'kr_opt': 50,      # Adjust marker size as needed
    'kt_opt': 43,
    'I_opt': 60,
    'E_opt': 35,
    'EI_opt': 60
}

# Set up the plot
plt.figure(figsize=(9, 6))

# Create scatter plot for each column with thin black edges
for col in columns:
    plt.scatter(
        [col] * len(test2_x),
        test2_x[col],
        label=col,
        color=colors[col],
        marker=markers[col],
        edgecolor='black',
        linewidth=0.5,
        s=marker_sizes[col],
        zorder=2
    )

# Connect the dots for each row
for i in range(len(test2_x)):
    plt.plot(columns, test2_x.iloc[i][columns], color='dimgrey', linestyle='-', linewidth
        =0.5, zorder=1)

# Add labels and title
plt.xlabel('Optimized updating parameters in x-direction', fontsize=12)
plt.ylabel('Scalar (-)', fontsize=12)
plt.xticks(rotation=45)

# Customize grid
plt.grid(True, which='major', axis='x', linestyle='-', linewidth=0.5, color='black')  # Major
     vertical grid lines

# Hide major horizontal grid lines
plt.grid(True, which='major', axis='y', linestyle='-', linewidth=0.5, color='white')

# Add ticks where vertical grid lines intersect horizontal lines
for col in columns:
    y_vals = [0.1, 1, 10]
    for y_val in y_vals:
        plt.scatter(col, y_val, color='black', marker='_', linewidths=1.7, s=40, zorder=0)

for col in columns:
    y_vals = [0.2,0.3,0.4, 0.5,0.6,0.7,0.8,0.9, 2,3,4,5,6,7,8,9]
    for y_val in y_vals:
        plt.scatter(col, y_val, color='black', marker='_', linewidths=0.5, s=2, zorder=0)

# Adjust spines
plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('white')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('white')
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('white')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.gca().yaxis.set_ticks_position('left') # Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=5, width=1, colors='black')
     # Customize tick parameters
# Set y-scale and limits
plt.yscale('log')
```

```
plt.ylim(0.1, 10)
plt.gca().set_yticks([0.1, 1, 10])
plt.gca().set_yticklabels(['0.1', '1', '10'])
#plt.tick_params(axis='y', which='major', direction='inout', length=6, width=1, colors='black')
#plt.gca().yaxis.grid(True, which='minor', linestyle='-', linewidth=0.25)
# Show legend
plt.legend(columns, bbox_to_anchor=(0.05, 1), loc='upper left', prop={'family': 'Arial', 'size': 10})

# Display the plot
plt.tight_layout()
plt.show()

# Dimension Parameters:
L = 160.58 #[m]
Lt = np.array([5.735, 11.47, 26.640, 89.91, 26.825]) #[m]
B = 28 #[m]
D = 28 #[m]
A = B * D #[m2]
#I_x = np.array([619.21, 948.47, 2727.22, 2335.82, 905.52]) #[m4]
#I_y = np.array([1536.25, 959.01, 1245.46, 1202.07, 831.57]) #[m4]
I_yy = np.array([1444,1500,1393,1389,1293]) #[m4]
I_xx = np.array([1532,1548,2796,2324,760]) #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([1933, 495, 486, 440, 383]) #[kg/m3]
E = np.array([38.2e9, 38.2e9, 38.2e9, 38.2e9, 32.8e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v)) #[N/m^2]
k = 0.85 #[-]

# Springs:
kr_x = 2380e9 #[Nm/rad]
kr_y = 2625e9 #[Nm/rad]
kt_y = 19e9 #[N/m]
kt_x = 4e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")

scaler = {"kr":0.4, "kt":9.0, "I":1.5, "E":7.0} #The parameters chosen are from the conclusion of sensitivity study 2
zz_trgt = np.array([57.165, 120.435, 153.735])  #The three locations where there are measurements taken of the New Orleans
z_trgt = zz_trgt/L

#Startpoints
startpoints = 400

fny = np.array([0.291, 1.332])
phiyy = [
    [-0.29, -0.90],
    [-0.78, 0.19],
    [-1, 1]
]
phiy = np.array(phiyy)

# Procedure

def adjust_modes(phi1, phi2):
    rmse_plus = np.sqrt(np.mean((phi1 - phi2) ** 2))
    rmse_minus = np.sqrt(np.mean((phi1 + phi2) ** 2))
    flip = np.ones_like(phi2)
    flip[rmse_plus > rmse_minus] = -1
    phi_flip = phi1 * flip
    return phi_flip

def modal_assurance_criterion(phi1, phi2):
    phi2 = adjust_modes(phi2, phi1)
```

```
    mac = np.abs(np.conj(phi1).dot(phi2))**2 / \
              (np.conj(phi1).dot(phi1) * np.conj(phi2).dot(phi2))
    return mac

timoshenko_y = []
for iy in range(len(H)):
    timoshenko_y.append(Timo(params={"E": E[iy], "I": I_yy[iy],"L": L,"rho": rho[iy], "A": A
        , "kG": k * G[iy], "kr": kr_y, "kt": kt_y}, height=H[iy]))
model_y = TimoPieceWiseBeam1D(beams=timoshenko_y, bc = bc)

y_hat_y = {"z": z_trgt, "fn": deepcopy(fny), "modes": deepcopy(phiy)}
parameters_to_update_y = {p:[1/10, 10] for p in list(scaler.keys())}
logscale = False

results_y = update_model(
    model=model_y,
    cost_func=cost_func1D,
    parameters_to_update=parameters_to_update_y,
    y_hat=y_hat_y,
    npts=startpoints,
    sim_type=UpdatingMethod.CMC,
    verbose=False,
    logscale=logscale
)
parameters_columns = [cc for cc in results_y.columns if "_st" in cc or "_opt" in cc]
plot_cols = [a + "_opt" for a in parameters_to_update_y.keys()]
map_opt2name = {pcl: k for k, pcl in zip(parameters_to_update_y.keys(), plot_cols)}
df_opt = results_y.copy()
df_opt.dropna(inplace=True)

if logscale:
    df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

# Results

t_y = results_y.sort_values(by='Convergence').head(startpoints)

quantile_threshold = t_y['Convergence'].quantile(0.125)
test_y = t_y[t_y['Convergence'] <= quantile_threshold]
print(test_y)

# Find the minimum and maximum values in each _opt column
min_values = test_y[[col for col in test_y.columns if col.endswith('_opt')]].min()
max_values = test_y[[col for col in test_y.columns if col.endswith('_opt')]].max()

# Update the boundaries
new_boundaries = {
    'kr': [min_values['kr_opt'], max_values['kr_opt']],
    'kt': [min_values['kt_opt'], max_values['kt_opt']],
    'I': [min_values['I_opt'], max_values['I_opt']],
    'E': [min_values['E_opt'], max_values['E_opt']]
}
print("New␣Boundaries:")
print(new_boundaries)

results_repeated2_y = update_model(
    model=model_y,
    cost_func=cost_func1D,
    parameters_to_update=new_boundaries,
    y_hat=y_hat_y,
    npts=startpoints,
    sim_type=UpdatingMethod.CMC,
    verbose=False,
    logscale=logscale
)
parameters_columns = [cc for cc in results_repeated2_y.columns if "_st" in cc or "_opt" in cc
    ]
plot_cols = [a + "_opt" for a in new_boundaries.keys()]
map_opt2name = {pcl: k for k, pcl in zip(new_boundaries.keys(), plot_cols)}
df_opt = results_repeated2_y.copy()
df_opt.dropna(inplace=True)
```

```
if logscale:
    df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

t2_y = results_repeated2_y.sort_values(by='Convergence').head(startpoints)
print(t2_y)

quantile_threshold = t2_y['Convergence'].quantile(0.125)
test2_y = t2_y[t2_y['Convergence'] <= quantile_threshold]
print(test2_y)

results_output_y = []
for ii in range(len(test2_y)):
    scaler_y = {
        "kr": test2_y["kr_opt"].iloc[ii],
        "kt": test2_y["kt_opt"].iloc[ii],
        "I": test2_y["I_opt"].iloc[ii],
        "E": test2_y["E_opt"].iloc[ii]
    }
    model_y.scale_parms(scalers=scaler_y)
    fn_y, phi_y, pts_y = model_y.modeshapes(pts=z_trgt, n_modes=2)

    mac_0 = modal_assurance_criterion(phi_y[:, 0], phiy[:, 0])
    mac_1 = modal_assurance_criterion(phi_y[:, 1], phiy[:, 1])

    results_output_y.append({
        "index": ii,
        "fn_y_1": fn_y[0],
        "fn_y_2": fn_y[1],
        "mac_1": mac_0,
        "mac_2": mac_1,
    })

results_df = pd.DataFrame(results_output_y)
stats_results = {}
z_value = 1.645

for col in ['fn_y_1', 'fn_y_2', 'mac_1', 'mac_2']:
    mean_val = results_df[col].mean()
    median_val = results_df[col].median()
    max_val = results_df[col].max()
    min_val = results_df[col].min()

    stats_results[col] = {
        "Median": median_val,
        "Mean": mean_val,
        "lower_bound": min_val,
        "upper_bound": max_val
    }

summary_df = pd.DataFrame(columns=["Measured fny", "Median", "Mean", "Lower Bound", "Upper Bound"])
for param, values in stats_results.items():
    if "fn_y_1" in param:
        measured_fny = fny[0]
    elif "fn_y_2" in param:
        measured_fny = fny[1]
    elif "fn_y_3" in param:
        measured_fny = fny[2]
    else:
        measured_fny = np.nan

    summary_df.loc[param] = [measured_fny, values["Median"], values["Mean"], values["lower_bound"], values["upper_bound"]]
print(summary_df)

def calculate_summary_stats(df, columns, direction):
    print(f"{direction}-direction")

    kr_summary_stats = []
    kt_summary_stats = []
```

```
    for column in columns:
        if "kr_opt" in column:
            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_boundkr = df[column].min()
            upper_boundkr = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            kr_summary_stats.append(pd.DataFrame({
                'Design': 1,
                'Median': median_val,
                'Lower Bound': lower_boundkr,
                'Upper Bound': upper_boundkr,
                'COV (%)': cv
            }, index=["kr_opt"]))

        elif "kt_opt" in column:
            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_boundkt = df[column].min()
            upper_boundkt = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            kt_summary_stats.append(pd.DataFrame({
                'Design': 1,
                'Median': median_val,
                'Lower Bound': lower_boundkt,
                'Upper Bound': upper_boundkt,
                'COV (%)': cv
            }, index=["kt_opt"]))

    kr_summary_stats_df = pd.concat(kr_summary_stats, axis=0)
    kt_summary_stats_df = pd.concat(kt_summary_stats, axis=0)

    print("Summary statistics for boundary conditions:")
    print(kr_summary_stats_df)
    print(kt_summary_stats_df.to_string(header=None))

    for ii in range(len(E)):
        print("")
        print(f"Summary statistics for discrete element {ii}")
        summary_stats_list = []

        for column in columns:
            if "kr_opt" in column or "kt_opt" in column:
                continue

            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_bound = df[column].min()
            upper_bound = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            if 'y' in direction:
                if "E_opt" in column:
```

```
                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val,
                        'Lower␣Bound': lower_bound ,
                        'Upper␣Bound': upper_bound,
                        'COV␣(%)': cv
                    }, index=["E_opt"])
                    summary_stats_list.append(summary_stats)
                elif "I_opt" in column:
                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val,
                        'Lower␣Bound': lower_bound ,
                        'Upper␣Bound': upper_bound,
                        'COV␣(%)': cv
                    }, index=["I_opt"])
                    summary_stats_list.append(summary_stats)
                    new_column = df["E_opt"] * df["I_opt"]
                    mean_val_new = new_column.mean()
                    median_val_new = new_column.median()
                    std_dev_new = new_column.std()

                    lower_bound_new = new_column.min()
                    upper_bound_new = new_column.max()

                    cv_new = round((std_dev_new / mean_val_new) * 100, 2)

                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val_new,
                        'Lower␣Bound': lower_bound_new ,
                        'Upper␣Bound': upper_bound_new,
                        'COV␣(%)': cv_new
                    }, index=["EI_opt"])
                    summary_stats_list.append(summary_stats)

        summary_stats_df = pd.concat(summary_stats_list, axis=0)
        print(summary_stats_df)

parameters_columns_y = [cc for cc in test2_y.columns if "_opt" in cc]
summary_stats_y = calculate_summary_stats(test2_y, parameters_columns_y, "y")

import matplotlib.pyplot as plt

# Assuming test2_y is already defined and has the columns 'kr_opt', 'kt_opt', 'I_opt', 'E_opt', and 'EI_opt'

# Add a new column 'EI_opt' which is the product of 'E_opt' and 'I_opt'
test2_y['EI_opt'] = test2_y['E_opt'] * test2_y['I_opt']

# Columns to plot
columns = ['kr_opt', 'kt_opt', 'E_opt', 'I_opt', 'EI_opt']

# Define colors for each column
colors = {
    'kr_opt': 'steelblue',
    'kt_opt': 'lightsteelblue',
    'I_opt': 'firebrick',
    'E_opt': 'goldenrod',
    'EI_opt': 'white'
}

# Define marker styles for each column
markers = {
    'kr_opt': 'o',     # circle
    'kt_opt': 's',     # square
    'I_opt': '^',      # triangle up
    'E_opt': 'D',      # diamond
    'EI_opt': 'v'      # plus (filled)
}

```

```python
# Define marker sizes for each column
marker_sizes = {
    'kr_opt': 50,      # Adjust marker size as needed
    'kt_opt': 43,
    'I_opt': 60,
    'E_opt': 35,
    'EI_opt': 60
}

# Set up the plot
plt.figure(figsize=(9, 6))

# Create scatter plot for each column with thin black edges
for col in columns:
    plt.scatter(
        [col] * len(test2_y),
        test2_y[col],
        label=col,
        color=colors[col],
        marker=markers[col],
        edgecolor='black',
        linewidth=0.5,
        s=marker_sizes[col],
        zorder=2
    )

# Connect the dots for each row
for i in range(len(test2_y)):
    plt.plot(columns, test2_y.iloc[i][columns], color='dimgrey', linestyle='-', linewidth=0.5, zorder=1)

# Add labels and title
plt.xlabel('Optimized updating parameters in y-direction', fontsize=12)
plt.ylabel('Scalar (-)', fontsize=12)
plt.xticks(rotation=45)

# Customize grid
plt.grid(True, which='major', axis='x', linestyle='-', linewidth=0.5, color='black')  # Major vertical grid lines

# Hide major horizontal grid lines
plt.grid(True, which='major', axis='y', linestyle='-', linewidth=0.5, color='white')

# Add ticks where vertical grid lines intersect horizontal lines
for col in columns:
    y_vals = [0.1, 1, 10]
    for y_val in y_vals:
        plt.scatter(col, y_val, color='black', marker='_', linewidths=1.7, s=40, zorder=0)

for col in columns:
    y_vals = [0.2,0.3,0.4, 0.5,0.6,0.7,0.8,0.9, 2,3,4,5,6,7,8,9]
    for y_val in y_vals:
        plt.scatter(col, y_val, color='black', marker='_', linewidths=0.5, s=2, zorder=0)

# Adjust spines
plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('white')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('white')
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('white')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.gca().yaxis.set_ticks_position('left') # Ensure ticks are on the left side of the plot
plt.tick_params(axis='y', which='major', direction='out', length=5, width=1, colors='black')  # Customize tick parameters
# Set y-scale and limits
plt.yscale('log')
plt.ylim(0.1, 10)
plt.gca().set_yticks([0.1, 1, 10])
plt.gca().set_yticklabels(['0.1', '1', '10'])
```

```
#plt.tick_params(axis='y', which='major', direction='inout', length=6, width=1, colors='black
    ')
#plt.gca().yaxis.grid(True, which='minor', linestyle='-', linewidth=0.25)
# Show legend
plt.legend(columns, bbox_to_anchor=(0.05, 1), loc='upper left', prop={'family': 'Arial', '
    size': 10})

# Display the plot
plt.tight_layout()
plt.show()

# Dimension Parameters:
L = 160.58 #[m]
Lt = np.array([5.735, 11.47, 26.640, 89.91, 26.825]) #[m]
B = 28 #[m]
D = 28 #[m]
A = B * D #[m2]
#I_x = np.array([619.21, 948.47, 2727.22, 2335.82, 905.52]) #[m4]
#I_y = np.array([1536.25, 959.01, 1245.46, 1202.07, 831.57]) #[m4]
I_yy = np.array([1444,1500,1393,1389,1293]) #[m4]
I_xx = np.array([1532,1548,2796,2324,760]) #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([1933, 495, 486, 440, 383]) #[kg/m3]
E = np.array([38.2e9, 38.2e9, 38.2e9, 38.2e9, 32.8e9]) #[N/m2]
v = 0.2 #[-]
G = E/(2*(1+v)) #[N/m^2]
k = 0.85 #[-]

# Springs:
kr_x = 2380e9 #[Nm/rad]
kr_y = 2625e9 #[Nm/rad]
kt_y = 19e9 #[N/m]
kt_x = 4e9 #[N/m]

#boudary conditions:
bc = ("rot+transl", "free")

scaler = {"kr":0.4, "kt":9.0, "I":1.5, "E":7.0} #The parameters chosen are from the
    conclusion of sensitivity study 2
zz_trgt = np.array([57.165, 120.435, 153.735])  #The three locations where there are
    measurements taken of the New Orleans
z_trgt = zz_trgt/L

#Startpoints
startpoints = 400

fny = np.array([0.291, 1.332, 2.771])
phiyy = [
    [-0.29, -0.90, -0.73],
    [-0.78, 0.19, 0.83],
    [-1, 1,-1]
]
phiy = np.array(phiyy)

# Procedure

def adjust_modes(phi1, phi2):
    rmse_plus = np.sqrt(np.mean((phi1 - phi2) ** 2))
    rmse_minus = np.sqrt(np.mean((phi1 + phi2) ** 2))
    flip = np.ones_like(phi2)
    flip[rmse_plus > rmse_minus] = -1
    phi_flip = phi1 * flip
    return phi_flip

def modal_assurance_criterion(phi1, phi2):
    phi2 = adjust_modes(phi2, phi1)
    mac = np.abs(np.conj(phi1).dot(phi2))**2 / \
              (np.conj(phi1).dot(phi1) * np.conj(phi2).dot(phi2))
    return mac
```

```
timoshenko_y = []
for iy in range(len(H)):
    timoshenko_y.append(Timo(params={"E": E[iy], "I": I_yy[iy],"L": L,"rho": rho[iy], "A": A
        , "kG": k * G[iy], "kr": kr_y, "kt": kt_y}, height=H[iy]))
model_y = TimoPieceWiseBeam1D(beams=timoshenko_y, bc = bc)

y_hat_y = {"z": z_trgt, "fn": deepcopy(fny), "modes": deepcopy(phiy)}
parameters_to_update_y = {p:[1/10, 10] for p in list(scaler.keys())}
logscale = False

results_y = update_model(
    model=model_y,
    cost_func=cost_func1D,
    parameters_to_update=parameters_to_update_y,
    y_hat=y_hat_y,
    npts=startpoints,
    sim_type=UpdatingMethod.CMC,
    verbose=False,
    logscale=logscale
)
parameters_columns = [cc for cc in results_y.columns if "_st" in cc or "_opt" in cc]
plot_cols = [a + "_opt" for a in parameters_to_update_y.keys()]
map_opt2name = {pcl: k for k, pcl in zip(parameters_to_update_y.keys(), plot_cols)}
df_opt = results_y.copy()
df_opt.dropna(inplace=True)

if logscale:
    df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)

# Results

t_y = results_y.sort_values(by='Convergence').head(startpoints)

quantile_threshold = t_y['Convergence'].quantile(0.125)
test_y = t_y[t_y['Convergence'] <= quantile_threshold]
print(test_y)

# Find the minimum and maximum values in each _opt column
min_values = test_y[[col for col in test_y.columns if col.endswith('_opt')]].min()
max_values = test_y[[col for col in test_y.columns if col.endswith('_opt')]].max()

# Update the boundaries
new_boundaries = {
    'kr': [min_values['kr_opt'], max_values['kr_opt']],
    'kt': [min_values['kt_opt'], max_values['kt_opt']],
    'I': [min_values['I_opt'], max_values['I_opt']],
    'E': [min_values['E_opt'], max_values['E_opt']]
}
print("New␣Boundaries:")
print(new_boundaries)

results_repeated2_y = update_model(
    model=model_y,
    cost_func=cost_func1D,
    parameters_to_update=new_boundaries,
    y_hat=y_hat_y,
    npts=startpoints,
    sim_type=UpdatingMethod.CMC,
    verbose=False,
    logscale=logscale
)
parameters_columns = [cc for cc in results_repeated2_y.columns if "_st" in cc or "_opt" in cc
    ]
plot_cols = [a + "_opt" for a in new_boundaries.keys()]
map_opt2name = {pcl: k for k, pcl in zip(new_boundaries.keys(), plot_cols)}
df_opt = results_repeated2_y.copy()
df_opt.dropna(inplace=True)

if logscale:
    df_opt[parameters_columns] = df_opt[parameters_columns].apply(np.exp)
```

```
t2_y = results_repeated2_y.sort_values(by='Convergence').head(startpoints)
print(t2_y)

quantile_threshold = t2_y['Convergence'].quantile(0.125)
test2_y = t2_y[t2_y['Convergence'] <= quantile_threshold]
print(test2_y)

results_output_y = []
for ii in range(len(test2_y)):
    scaler_y = {
        "kr": test2_y["kr_opt"].iloc[ii],
        "kt": test2_y["kt_opt"].iloc[ii],
        "I": test2_y["I_opt"].iloc[ii],
        "E": test2_y["E_opt"].iloc[ii]
    }
    model_y.scale_parms(scalers=scaler_y)
    fn_y, phi_y, pts_y = model_y.modeshapes(pts=z_trgt, n_modes=3)

    mac_0 = modal_assurance_criterion(phi_y[:, 0], phiy[:, 0])
    mac_1 = modal_assurance_criterion(phi_y[:, 1], phiy[:, 1])
    mac_2 = modal_assurance_criterion(phi_y[:, 2], phiy[:, 2])

    results_output_y.append({
        "index": ii,
        "fn_y_1": fn_y[0],
        "fn_y_2": fn_y[1],
        "fn_y_3": fn_y[2],
        "mac_1": mac_0,
        "mac_2": mac_1,
        "mac_3": mac_2
    })

results_df = pd.DataFrame(results_output_y)
stats_results = {}
z_value = 1.645

for col in ['fn_y_1', 'fn_y_2','fn_y_3', 'mac_1', 'mac_2', 'mac_3']:
    mean_val = results_df[col].mean()
    median_val = results_df[col].median()
    max_val = results_df[col].max()
    min_val = results_df[col].min()

    stats_results[col] = {
        "Median": median_val,
        "Mean": mean_val,
        "lower_bound": min_val,
        "upper_bound": max_val
    }

summary_df = pd.DataFrame(columns=["Measured fny", "Median", "Mean", "Lower Bound", "Upper Bound"])
for param, values in stats_results.items():
    if "fn_y_1" in param:
        measured_fny = fny[0]
    elif "fn_y_2" in param:
        measured_fny = fny[1]
    elif "fn_y_3" in param:
        measured_fny = fny[2]
    else:
        measured_fny = np.nan

    summary_df.loc[param] = [measured_fny, values["Median"], values["Mean"], values["lower_bound"], values["upper_bound"]]
print(summary_df)

def calculate_summary_stats(df, columns, direction):
    print(f"{direction}-direction")

    kr_summary_stats = []
    kt_summary_stats = []
```

```
    for column in columns:
        if "kr_opt" in column:
            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_boundkr = df[column].min()
            upper_boundkr = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            kr_summary_stats.append(pd.DataFrame({
                'Design': 1,
                'Median': median_val,
                'Lower Bound': lower_boundkr,
                'Upper Bound': upper_boundkr,
                'COV (%)': cv
            }, index=["kr_opt"]))

        elif "kt_opt" in column:
            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_boundkt = df[column].min()
            upper_boundkt = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            kt_summary_stats.append(pd.DataFrame({
                'Design': 1,
                'Median': median_val,
                'Lower Bound': lower_boundkt,
                'Upper Bound': upper_boundkt,
                'COV (%)': cv
            }, index=["kt_opt"]))

    kr_summary_stats_df = pd.concat(kr_summary_stats, axis=0)
    kt_summary_stats_df = pd.concat(kt_summary_stats, axis=0)

    print("Summary statistics for boundary conditions:")
    print(kr_summary_stats_df)
    print(kt_summary_stats_df.to_string(header=None))

    for ii in range(len(E)):
        print("")
        print(f"Summary statistics for discrete element {ii}")
        summary_stats_list = []

        for column in columns:
            if "kr_opt" in column or "kt_opt" in column:
                continue

            mean_val = df[column].mean()
            median_val = df[column].median()
            sem_val = df[column].sem()
            std_dev = df[column].std()
            z_value = 1.645

            lower_bound = df[column].min()
            upper_bound = df[column].max()

            cv = round((std_dev / mean_val) * 100, 2)

            if 'y' in direction:
                if "E_opt" in column:
```

```
                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val,
                        'Lower␣Bound': lower_bound ,
                        'Upper␣Bound': upper_bound,
                        'COV␣(%)': cv
                    }, index=["E_opt"])
                    summary_stats_list.append(summary_stats)
                elif "I_opt" in column:
                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val,
                        'Lower␣Bound': lower_bound ,
                        'Upper␣Bound': upper_bound,
                        'COV␣(%)': cv
                    }, index=["I_opt"])
                    summary_stats_list.append(summary_stats)
                    new_column = df["E_opt"] * df["I_opt"]
                    mean_val_new = new_column.mean()
                    median_val_new = new_column.median()
                    std_dev_new = new_column.std()

                    lower_bound_new = new_column.min()
                    upper_bound_new = new_column.max()

                    cv_new = round((std_dev_new / mean_val_new) * 100, 2)

                    summary_stats = pd.DataFrame({
                        'Design': 1,
                        'Median': median_val_new,
                        'Lower␣Bound': lower_bound_new ,
                        'Upper␣Bound': upper_bound_new,
                        'COV␣(%)': cv_new
                    }, index=["EI_opt"])
                    summary_stats_list.append(summary_stats)

        summary_stats_df = pd.concat(summary_stats_list, axis=0)
        print(summary_stats_df)

parameters_columns_y = [cc for cc in test2_y.columns if "_opt" in cc]
summary_stats_y = calculate_summary_stats(test2_y, parameters_columns_y, "y")
```

Appendix Timoshenko beam model

```
%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko import Timo, TimoPieceWiseBeam1D, TimoPieceWiseBeam2D, UpdatingMethod, cost_func1D, update_model
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
## Parameters

# Dimension Parameters:
L = 2 #[m]
Lt = np.array([1,1]) #[m]
B = 0.1 #[m]
D = 0.1 #[m]

A = B * D #[m2]
I = 1/12 * B * D**3 #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([8000,8000]) #[kg/m3]
E = np.array([260e9,260e9]) #[N/m2]
G = np.array([100e9,100e9]) #[N/m2]
k = 5/6 #[-]
```

```

#boudary conditions:
bc = ("hinged", "hinged")

## Procedure

timoshenko = []
for ix in range(len(H)):
    timoshenko.append(Timo(params = {"E": E[ix], "I": I, "L": L, "rho": rho[ix], "A": A, "kG"
        : k*G[ix]},height= H[ix]))
pb = TimoPieceWiseBeam1D(beams=timoshenko, bc=bc)
fn, phi, pts = pb.modeshapes(max_fn=30000, n_intervals=10000, n_modes= 50)

## Results

print("The first 50 natural frequencies (in rad/s) of the Timoshenko beam are:")
for ii in range(50):
    print(f"Mode {ii + 1} :", fn[ii] * 2 * np.pi)

plt.figure(figsize=(6, 6))
plt.plot(pts, phi[:,0], color='blue', label='Mode 1')
plt.plot(pts, phi[:,1], '--', color='red', label='Mode 2')
plt.plot(pts, -phi[:,2], '-.', color='green', label='Mode 3')
plt.plot(pts, -phi[:,3], '--', color='magenta', label='Mode 4')
plt.plot(pts, phi[:,4], '-.', color='black', label='Mode 5')

plt.ylim(-2, 2)


plt.ylabel('Normilized displacement (-)')
plt.xlabel('Normalized length (-)')
#plt.title('The mode shape of mode 1,2,3,4,and 5')
plt.xlim(0,1)
plt.yticks([-2,-1.6, -1.2, -0.8, -0.4, 0, 0.4, 0.8, 1.2, 1.6, 2.0])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.tick_params(axis='y', which='both', direction='inout', length=6, width=1, colors='black')
plt.legend(frameon=True, loc='lower left')
plt.grid(True)
plt.show()

plt.figure(figsize=(6, 6))
plt.plot(pts, -phi[:,25], color='blue', label='Mode 26')
plt.ylim(-0.5, 0.5)
plt.xlim(0,1)
plt.ylabel('Nodal displacement (-)')
plt.xlabel('Normalized length (-)')
#plt.title('The mode shape of mode 26 (at transition frequency $\omega_c$)')
plt.yticks([-0.5,-0.4, -0.3, -0.2, -0.1, 0, 0.1, 0.2, 0.3, 0.4, 0.5])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.legend(frameon=True, loc='lower left')
plt.grid(True)
plt.show()

plt.figure(figsize=(6, 6))
plt.plot(pts, phi[:,26], color='blue', label='Mode 27')
plt.plot(pts, phi[:,27], '--', color='red', label='Mode 28')
plt.plot(pts, -phi[:,29], '--', color='magenta', label='Mode 30')
plt.plot(pts, -phi[:,30], '-.', color='black', label='Mode 31')

plt.ylim(-2, 2)

plt.ylabel('Nodal displacement (-)')
plt.xlabel('Normalized length (-)')
#plt.title('The mode shape of mode 27, 28, 30 and 31')
plt.xlim(0,1)
plt.yticks([-2,-1.6, -1.2, -0.8, -0.4, 0, 0.4, 0.8, 1.2, 1.6, 2.0])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.legend(frameon=True, loc='lower left')
plt.grid(True)
plt.show()

```

```
plt.figure(figsize=(6, 6))
plt.plot(pts, phi[:,28], color='blue', label='Mode 29')
plt.plot(pts, -phi[:,31], '--', color='red', label='Mode 32')

plt.ylim(-2, 2)
plt.xlim(0,1)
plt.yticks([-2,-1.6, -1.2, -0.8, -0.4, 0, 0.4, 0.8, 1.2, 1.6, 2.0])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])

plt.ylabel('Nodal displacement (-)')
plt.xlabel('Normalized length (-)')
#plt.title('The mode shapes of modes 29 and 32')
plt.legend(frameon=True, loc='lower left')
plt.tick_params(axis='y', which='major', direction='inout', length=6, width=1, colors='black')
plt.tick_params(axis='y', which='minor', direction='inout', length=4, width=0.5, colors='gray')
plt.grid(True)
plt.show()

import matplotlib.pyplot as plt


# Plotting the data
plt.figure(figsize=(6, 6))
plt.plot(pts, phi[:,28], color='blue', label='Mode 29')
plt.plot(pts, -phi[:,31], '--', color='red', label='Mode 32')

plt.ylim(-2, 2)
plt.xlim(0, 1)
plt.yticks([-2, -1.6, -1.2, -0.8, -0.4, 0, 0.4, 0.8, 1.2, 1.6, 2.0])
plt.xticks([0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])

plt.ylabel('Nodal displacement (-)')
plt.xlabel('Normalized length (-)')
plt.legend(frameon=True, loc='lower left')
plt.grid(True)

# Customizing tick parameters for y-axis only
plt.tick_params(axis='y', which='major', direction='inout', length=6, width=1, colors='black')
plt.tick_params(axis='y', which='minor', direction='inout', length=4, width=0.5, colors='gray')

plt.show()


import matplotlib.pyplot as plt
import numpy as np

# Generate some sample data
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)

# Plotting the data
plt.figure(figsize=(8, 6))
plt.plot(x, y, color='blue', label='sin(x)')

plt.ylim(-1.2, 1.2)
plt.xlim(0, 2*np.pi)
plt.yticks(np.linspace(-1, 1, 5))  # Set y-axis ticks at specific intervals

plt.xlabel('x')
plt.ylabel('sin(x)')
plt.title('Sine Function')

plt.legend()

# Customizing y-axis tick parameters
plt.tick_params(axis='y', which='major', direction='inout', length=6, width=1, colors='black')
```

```
plt.show()

%load_ext autoreload
%autoreload 2
from copy import deepcopy
from pyvibe.models.Timoshenko_phi import Timo, TimoPieceWiseBeam1D, TimoPieceWiseBeam2D, UpdatingMethod, cost_func1D, update_model
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
## Parameters

# Dimension Parameters:
L = 2 #[m]
Lt = np.array([1,1]) #[m]
B = 0.1 #[m]
D = 0.1 #[m]

A = B * D #[m2]
I = 1/12 * B * D**3 #[m4]
H = Lt/L #[-]

# Material properties:
rho = np.array([8000,8000]) #[kg/m3]
E = np.array([260e9,260e9]) #[N/m2]
G = np.array([100e9,100e9]) #[N/m2]
k = 5/6 #[-]

#boudary conditions:
bc = ("hinged", "hinged")
## Procedure

timoshenko = []
for ix in range(len(H)):
    timoshenko.append(Timo(params = {"E": E[ix], "I": I, "L": L, "rho": rho[ix], "A": A, "kG": k*G[ix]},height= H[ix]))
pb = TimoPieceWiseBeam1D(beams=timoshenko, bc=bc)
fn, phi, pts = pb.modeshapes(max_fn=30000, n_intervals=10000, n_modes= 50)
## Results

plt.figure(figsize=(6, 6))
plt.plot(pts, phi[:,0], color='blue', label='Mode␣1')
plt.plot(pts, -phi[:,1], '--', color='red', label='Mode␣2')
plt.plot(pts, phi[:,2], '-.', color='green', label='Mode␣3')
plt.plot(pts, -phi[:,3], '--', color='magenta', label='Mode␣4')
plt.plot(pts, -phi[:,4], '-.', color='black', label='Mode␣5')

#plt.ylim(-2, 2)
plt.ylabel('Normalized␣rotation␣(-)')
plt.xlabel('Normalized␣length␣(-)')
#plt.title('The mode shape of mode 1,2,3,4,and 5')
plt.xlim(0,1)
plt.yticks([-2,-1.6, -1.2, -0.8, -0.4, 0, 0.4, 0.8, 1.2, 1.6, 2.0])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.tick_params(axis='y', which='both', left=True, right=False, direction='out', length=8, width=0.5)
plt.tick_params(axis='x', which='both', bottom=True, top=False, direction='out', length=8, width=0.5)
plt.legend(frameon=True, loc='lower␣left')
plt.grid(True)
plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('black')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('black')
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('black')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.show()
```

```
plt.figure(figsize=(6, 6))
plt.plot(pts, phi[:,25], color='blue', label='Mode 26')
plt.xlim(0, 1)
plt.ylim(-1.2,1.2)
plt.ylabel('Normalized rotation (-)')
plt.xlabel('Normalized length (-)')
#plt.title('The mode shape of mode 26 (at transition frequency $\omega_c$)')
plt.yticks([-1.2,-1.0,-0.8,-0.6,-0.4,-0.2,0.0,0.2,0.4,0.6,0.8,1.0,1.2])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.tick_params(axis='y', which='both', left=True, right=False, direction='out', length=8,
    width=0.5)
plt.tick_params(axis='x', which='both', bottom=True, top=False, direction='out', length=8,
    width=0.5)
plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('black')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('black')
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('black')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.legend(frameon=True, loc='lower left')
plt.grid(True)
plt.show()

plt.figure(figsize=(6, 6))
plt.plot(pts, phi[:,26], color='blue', label='Mode 27')
plt.plot(pts, phi[:,27], '--', color='red', label='Mode 28')
plt.plot(pts, phi[:,29], '--', color='magenta', label='Mode 30')
plt.plot(pts, phi[:,30], '-.', color='black', label='Mode 31')

plt.ylim(-2, 2)

plt.ylabel('Normalized rotation (-)')
plt.xlabel('Normalized length (-)')
#plt.title('The mode shape of mode 27, 28, 30 and 31')
plt.xlim(0,1)
plt.yticks([-2,-1.6, -1.2, -0.8, -0.4, 0, 0.4, 0.8, 1.2, 1.6, 2.0])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.tick_params(axis='y', which='both', left=True, right=False, direction='out', length=8,
    width=0.5)
plt.tick_params(axis='x', which='both', bottom=True, top=False, direction='out', length=8,
    width=0.5)
plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('black')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('black')
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('black')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.legend(frameon=True, loc='lower left')
plt.grid(True)
plt.show()

plt.figure(figsize=(6, 6))
plt.plot(pts, -phi[:,28], color='blue', label='Mode 29')
plt.plot(pts, phi[:,31], '--', color='red', label='Mode 32')

plt.ylim(-2, 2)
plt.xlim(0,1)
plt.yticks([-2,-1.6, -1.2, -0.8, -0.4, 0, 0.4, 0.8, 1.2, 1.6, 2.0])
plt.xticks([0,0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
plt.tick_params(axis='y', which='both', left=True, right=False, direction='out', length=8,
    width=0.5)
plt.tick_params(axis='x', which='both', bottom=True, top=False, direction='out', length=8,
    width=0.5)
plt.gca().spines['top'].set_linewidth(0.5)
plt.gca().spines['top'].set_color('black')
plt.gca().spines['right'].set_linewidth(0.5)
plt.gca().spines['right'].set_color('black')
```

```
plt.gca().spines['bottom'].set_linewidth(0.5)
plt.gca().spines['bottom'].set_color('black')
plt.gca().spines['left'].set_linewidth(0.5)
plt.gca().spines['left'].set_color('black')
plt.ylabel('Normalized rotation (-)')
plt.xlabel('Normalized length (-)')
#plt.title('The mode shapes of modes 29 and 32')
plt.legend(frameon=True, loc='lower left')
plt.grid(True)
plt.show()
```