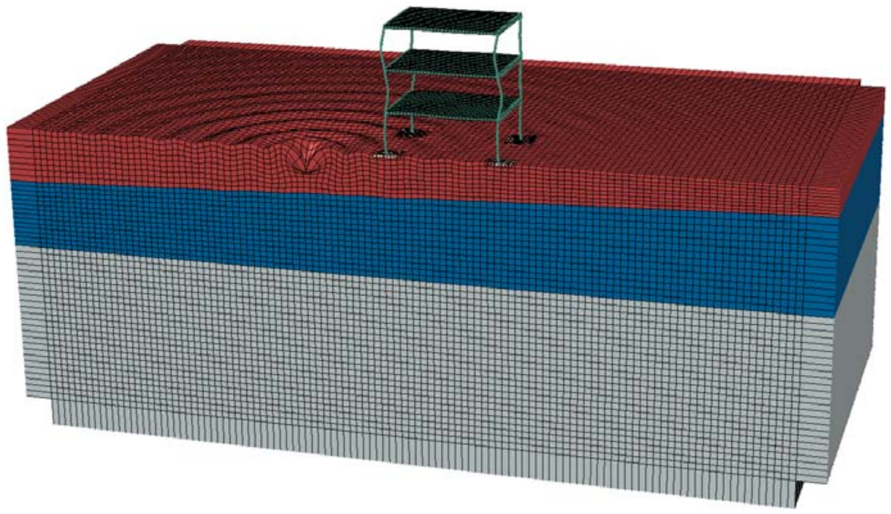
LUND UNIVERSITY



MODELLING GROUND VIBRATIONS IN ABAQUS USING PYTHON

ARGETA JUPOLLI

Structural Mechanics

Master's Dissertation

DEPARTMENT OF CONSTRUCTION SCIENCES

DIVISION OF STRUCTURAL MECHANICS

ISRN LUTVDG/TVSM--25/5278--SE (1-74) | ISSN 0281-6679

MASTER'S DISSERTATION

MODELLING GROUND VIBRATIONS IN ABAQUS USING PYTHON

ARGETA JUPOLLI

Supervisor: **LINUS ANDERSSON**, Assistant Professor, Division of Structural Mechanics, LTH.
Assistant Supervisors: Dr **JONAS LINDEMANN**, Division of Structural Mechanics, LTH | Lunarc
and **KHUONG AN UNG**, Doctoral Student, Division of Structural Mechanics, LTH.

Examiner: **PETER PERSSON**, Associate Professor, Division of Structural Mechanics, LTH.

Printed by V-husets tryckeri LTH, Lund, Sweden, July 2025 *(Pl)*.

For information, address:
Division of Structural Mechanics,
Faculty of Engineering LTH, Lund University, Box 118, SE-221 00 Lund, Sweden.
Homepage: www.byggmek.lth.se

Abstract

The need for accurate and computationally efficient methods for analysing the effects of ground vibrations on a structure is increasing. This is partly due to continued urbanization leading to structures being placed in closer proximity to vibrational sources, as well as increased environmental awareness leading to more lightweight structures, such as timber structures, being built. The purpose of this study was to create accurate and computationally efficient FE ground models for the analyses of ground vibrations in Abaqus.

This objective was achieved by creating tailored FE ground models based on the properties of the ground layers and the frequency interval that was to be studied. These parameters determined the longest and shortest wavelength of the vibrations, which in turn were used to determine the model size and element size respectively. P-waves have the longest wavelength and were used along with the lower frequency in a given range for determining model size, while Rayleigh wavelengths are the shortest and were used along with the upper frequency to determine element size.

A complete program was developed using Python and consists of the creation of axisymmetric and 3D FE models, creation of input files compatible with Abaqus, submission of jobs for analyses in Abaqus and extraction of results. Additionally, a user interface was created to increase user-friendliness. Parameter studies were conducted with the purpose of determining the appropriate number of P-wavelengths, the number of elements per Rayleigh wavelength, the size of the frequency increments and the size of the frequency bands that each model was to be tailored to. The resulting models were validated by comparing the results to a well-established semi-analytical model used for analyses of traffic-induced vibrations. The time saved through the use of the tailored models was determined by comparing analyses times for the tailored models with less tailored models.

The parameter study resulted in the tailored models consisting of 1.5 P-wavelengths used to determine model size, 5 elements per Rayleigh wavelength used to determine element size, 1 Hz sized increments being used in the analyses and each model being tailored to a 5 Hz frequency band. When compared to results from the semi-analytical model, the chosen values from the parameter studies resulted in sufficiently accurate results up to about 80 Hz, after which the results appeared to oscillate more. The accuracy above 80 Hz was deliberately compromised in favour of reasonable analyses times.

The 3D models were, even when tailored, too large for analyses on a regular computer and had to be submitted to a supercomputer for analyses. This was mostly due to the models' extensive RAM memory usage, often requiring 256 GB RAM memory and in some cases even 512 GB. This was an unexpected complication which resulted in the models, neither 3D nor axisymmetric, being able to be analysed in the 1-100 Hz range when not divided into smaller models, even on the supercomputer available for this project. In order to determine the computational efficiency of the tailored models, they were therefore compared to less tailored models that used the full computational capacity available for this project.

For sequential analyses, the tailored axisymmetric models were in the 1-100 Hz interval 4.4 times faster, taking 8 minutes instead of 36.5 minutes. The tailored 3D models were for sequential analyses in the 36-100 Hz interval 3.8 times faster, taking 2 days and 10 hours instead of 9 days and 4 hours for its less tailored counterparts. When applying parallel analyses, even more significant time savings were achieved. The tailored axisymmetric models were for the 1-100 Hz interval 12.2 times faster, taking 2.5 minutes instead of 30.5 minutes, while the tailored 3D models in the 75-100 Hz interval were 21 times faster, taking 4.5 hours instead of 4 days.

Acknowledgements

This dissertation concludes my five years of studies at Lund University. Looking back, these years have provided times of fun, times of less fun, times of enlightenment and times of frustration. However, they have without a doubt been my most instructive and rewarding years thus far.

A project like this can often feel daunting, such was the case for me at least. I am therefore very thankful to have had such great supervisors. Linus Andersson, whose knowledge of Abaqus and dynamics, but really of most things, was helpful throughout all stages of the project. Assistant supervisors Khuong An Ung, who made me feel comfortable asking all the stupid questions, and Jonas Lindemann, whose expertise in Python provided a big comfort. I would also like to thank my examiner Peter Persson for being a great teacher, proposing this project to me and providing valuable input. I am equally grateful to the whole Division of Structural Mechanics for providing a fruitful work environment with nice company and really strong coffee, which was appreciated at times when motivation was low.

To my loved ones, for all the different ways you have supported and helped me, thank you.

[illegible]

Argeta Jupolli

Lund, June 2025

Contents

1 Introduction

This project involves the development of a Python-based program intended to be used for Abaqus analysis of structures subjected to traffic-induced vibrations. The reader is presumed to have basic knowledge of the finite element (FE) method, the FE software Abaqus and programming with Python. In this chapter, the background as to why a program like this is beneficial will be explained, as well as the methodology used to develop it.

1.1 Background

The demand for efficient methods to analyse the effects of traffic-induced vibrations on buildings and structures is growing. This is partly driven by continued urbanization, which places buildings increasingly close to roads, railways, and other sources of vibration. Additionally, the construction industry's focus on sustainability has led to an increase in the use of environmentally friendly materials. However, lightweight structures, such as timber buildings, are often considered to be sensitive to traffic-induced noise and vibrations.

When using FE software like Abaqus to analyse ground vibrations, large ground models are required to accurately capture how vibrations propagate through the ground. These models are computationally expensive, often requiring days or even weeks to process.

One approach to improving computational efficiency is parallelizing analyses, where calculations are divided into smaller tasks that run simultaneously on multiple processors. However, another option is to tailor the models by adjusting the spatial discretization and the size of the ground domain in the numerical model based on the vibration frequencies to be studied. This would allow for different model sizes, each corresponding to a specific frequency range. At higher frequencies, shorter wavelengths reduce the required ground size, enabling smaller models that still yield accurate results.

1.2 Aim and objectives

The aim of this dissertation is to develop a Python-based program that generates efficient, ready-to-use ground models for analysis in Abaqus, focusing on the impact of traffic-induced vibrations on structures. The study will explore parameter selection impacting ground- and element size to ensure both accuracy and computational efficiency. To achieve this, the following objectives will be examined:

- How can accurate ground vibration be achieved using Python while also being compatible with Abaqus?
- Which parameters are relevant for accurate and computational efficient ground modelling?
- How much time can be saved in Abaqus analysis through the use of tailored models?

1.3 Methodology

The methodology consisted of a literature study as well as numerical modelling, numerical analysis and post-processing of results through the development of a program and a user interface.

The literature study was conducted focusing on relevant topics related to wave propagation in elastic solids, numerical modelling and methods for geometry and mesh generation.

Significant emphasis was placed on numerical modelling procedures, which were conducted using Python. The primary focus was on ensuring correct partitioning of the model to accurately represent different ground layers and non-reflective boundaries. Compatibility with the Abaqus framework for modelling was also a key consideration. Python-related concepts such as object-oriented programming and inheritance were applied. A user interface was developed using PyQt, a Python library that facilitates building user interfaces with the Qt framework, chosen for its simplicity and user-friendliness.

To ensure the validity of the tailored models, the numerical analysis conducted in Abaqus was compared to the results from a well-established semi-analytical approach used for analysing traffic-induced vibrations. This allowed for an investigation into which values for different parameters provided the most computationally efficient result while also being sufficiently accurate.

Finally, an example of an application for the program was presented by including a building and inclining the ground layers.

1.4 Limitations

- The work will be limited to creating ground models that are compatible with Abaqus.
- Only linear elastic material behaviour is considered.
- Only the 1-100 Hz interval is studied.

2 Governing theory

In this chapter, relevant theory for this project will be presented, starting with volume waves in elastic solids followed by numerical modelling and lastly methods for geometry and mesh generation.

2.1 Volume waves in elastic solids

Wave propagation in the ground is of interest as it affects both people and buildings. This is true for natural causes of vibrations, such as earthquakes, as well as man made causes, such as traffic-induced vibrations.

2.1.1 Wave types and their effect on ground modelling

There are three different types of waves that usually propagate through the ground due to some vibrational source. These are the P-wave, the S-wave and the Rayleigh wave, detailed descriptions of which are found in [1].

The P-wave, also known as the primary wave, gets its name from being the fastest of the three wave types and therefore the first to arrive at an observation point. It is a compressive wave, meaning its particle motion is parallel to the direction of wave propagation. The S-wave, i.e. the secondary wave, is slower and arrives at the observation point after the P-wave. The particle motion of the S-wave is perpendicular to the direction of propagation, making it a transversal wave, see Figure 2.1.

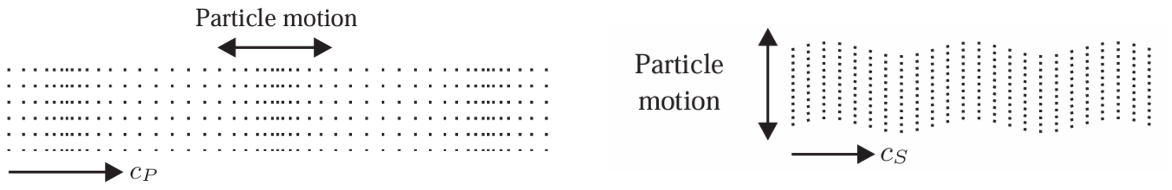


Figure 2.1: Particle motion of a plane P-wave, left, and plane S-wave, right [1].

The Rayleigh wave propagates in semi-infinite solid media, such as a ground material, and has an amplitude that decreases with increased distance from the surface. This wave is typically slower than the S-wave and contains both pressure and shear components.

The wave speeds are related to the material's Young's modulus and Poisson's ratio through Lamé constants λ and μ

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}. \tag{2.1}$$

From Lamé constants the P- and S- wave speeds can be determined as

$$c_P = \sqrt{\frac{\lambda + 2\mu}{\rho}}, \qquad c_S = \sqrt{\frac{\mu}{\rho}}. \tag{2.2}$$

The Rayleigh wave speed is uniquely defined for any Poisson's ratio and, for a half-sphere consisting of one ground material, increases from $c_R \approx 0.862 c_S$ for $\nu = 0$ to $c_R \approx 0.955 c_S$ for $\nu = 0.5$.

When the wave speed is known, the wavelength for a wave can be determined for a given frequency,

$$\lambda = \frac{c}{f}. \tag{2.3}$$

As the P-wave is the fastest of the three waves, its wavelength at a given frequency will be the longest of the three. In the same manner, the Rayleigh wavelength will be the shortest of the three.

When modelling a ground domain using the FE method, it is inevitable that the modelled geometry has boundaries, i.e. the spatial discretization must be applied to a bounded region. Although non-reflective boundaries can be modelled approximately using so-called infinite elements, see further Section 2.2.2, the model boundaries lead to reflections of the waves in the model that are not present in real life. In order to minimize these reflections, it is reasonable to assume that there should be some minimum distance between the boundaries of the ground domain and the loading and observation points. Furthermore, to consider the frequency dependency, such a minimum distance may be expressed in terms of a given number of wavelengths. With the P-wavelength being the longest, it may be appropriate that this should be the wavelength governing the size of the ground domain.

Another important aspect of FE modelling is the size of the finite elements. Similarly to the reasoning of the governing wavelength for the model size, one could define the element size by considering a certain number of elements per wavelength. As the Rayleigh wavelength is the shortest, this could be deemed most appropriate for determining the element size.

Given Equation 2.3, it can be concluded that when studying a certain frequency range, the lower frequency in the range will yield a longer wavelength and the upper frequency in the range will yield a shorter wavelength. Applying this to the logic of model- and element sizes, it follows that the model size should be dependent on the lower frequency in a range, and the element size on the upper frequency in a range, yielding the longest P-wavelength and shortest Rayleigh wavelength respectively.

$$\lambda_P = \frac{c_P}{f_{\text{lower}}}, \qquad \lambda_R = \frac{c_R}{f_{\text{upper}}} \tag{2.4}$$

Each frequency interval that is to be studied will have a given ratio between the P-wavelength, relating to model size, and the Rayleigh wavelength, relating to element size, according to:

$$\frac{\lambda_P}{\lambda_R} = \frac{f_{\text{upper}}}{f_{\text{lower}}} \cdot \frac{c_P}{c_R} \tag{2.5}$$

As c_P and c_R depend on the properties of the ground material, they remain constant throughout the frequency interval. The critical ratio is therefore between the upper and lower frequency in a chosen interval. For example, the frequency interval 1-5 Hz will give a ratio of 5 between the P-wavelength and Rayleigh wavelength, while the frequency interval 95-100 Hz, with a frequency band of the same size as previously, will give a ratio of $100/95 = 1.053$.

The ratio will increase linearly with the chosen number of P-wavelengths and number of elements per Rayleigh wavelength. For example, if 2 P-wavelengths is chosen for determining the model size and 5 elements per Rayleigh wavelength is chosen for determining element size, the ratio between model- and element size becomes

$$\frac{2\lambda_P}{\frac{\lambda_R}{5}} = 10 \cdot \frac{f_{\text{upper}}}{f_{\text{lower}}} \cdot \frac{c_P}{c_R}. \tag{2.6}$$

This suggests two things. First, for low frequencies, much smaller frequency bands are needed to maintain approximately the same ratio between model- and element size as for the larger frequencies. Second, the chosen number of P-wavelengths per model and elements per Rayleigh wavelength should be chosen with care as to not create unnecessarily large models with small elements that will be very computational expensive.

2.1.2 Damping of waves

At far distances from the source of a traffic-induced vibration, it has been observed that the Rayleigh wave leads to strong vibrations, whereas P- and S-waves vanish. This may be due to two main reasons. The majority of the energy transmitted to the ground by traffic leads to the generation of Rayleigh waves, i.e. surface waves. The other reason lies in the way the different waves spread. P- and S-waves spread over the volume creating spherical wavefronts, whereas the Rayleigh wave is bound to the surface and therefore spreads only through the surface and not the volume, see Figure 2.2. Since the P- and S-waves spread in three directions, a faster decay of energy occurs and therefore smaller amplitudes are achieved with increased distance. The decay of the amplitude of a wave due to spreading of the energy over a larger area or volume is called geometrical damping, i.e. the P- and S-wave exhibit larger geometrical damping than the Rayleigh wave at far distances.

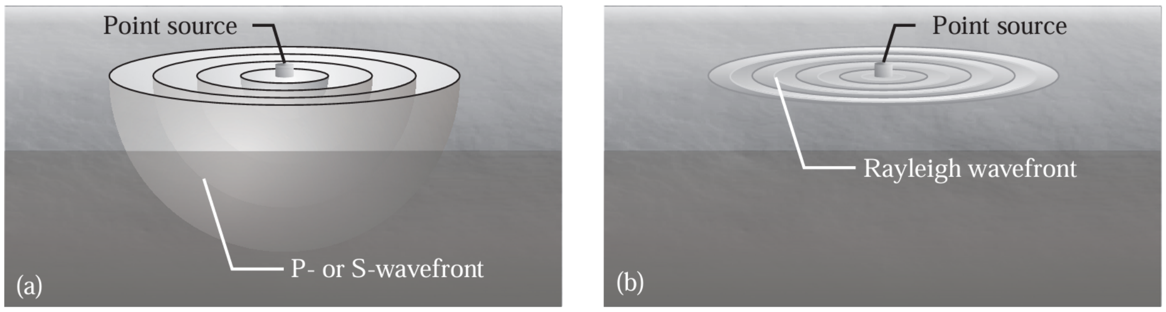


Figure 2.2: Geometrical damping from a point source on the ground surface for (a) P- and S-waves and (b) the Rayleigh wave [1].

Damping of waves also occurs through material damping, where mechanical energy is transformed into thermal energy, i.e. heat. This occurs in all real materials and is caused by mechanisms such as friction between particles or grains and leads to exponential decay of the displacement amplitudes over distance. This type of damping causes a shift in the wave's phase velocity, which leads the wavenumber to become complex. Material damping is represented by a loss factor η.

Geometrical damping is of greater importance than material damping in the near-field, whereas the reverse is true for the far-field. A more detailed explanation of the damping of waves can be found in [1].

2.1.3 Describing waves using complex notation

A wave can be described using complex notation, explained in detail in [2]. All analyses in this project are performed in the frequency domain, which naturally lends itself to complex notation representation of wave phenomena. This approach allows for efficient handling of both the amplitude and phase characteristics of waves, as well as the incorporation of material damping effects through complex material properties.

This is done in the form

$$z = x + iy = |z|(cos\varphi + isin\varphi) = |z|e^{i\varphi}, \tag{2.7}$$

where x is the real part of z and y is the imaginary part of z.

It is often of interest to study the magnitude, which refers to the total size or strength of the wave. The magnitude is defined by $|z|$ and is found from

$$|z| = \sqrt{x^2 + y^2}. \tag{2.8}$$

2.2 Numerical modelling

Physical problems are often complicated and difficult to analyse by experimental testing. For this purpose, numerical modelling is very useful. Numerical modelling is used to analyse complicated real life physical problems by means of mathematical models that describe the physical conditions of the problem. The analysis can be performed using computational simulation which simulates physical reality.

There have been several numerical modelling methods proposed for modelling ground-structure interaction. One of these is the Thomson-Haskell model, which is a semi-analytical method used for the analysis of layered half-spaces subjected to moving or stationary loads, in other words a method that may be used for analysing traffic-induced vibrations. This approach is computationally efficient as it, in contrast to the FE method, avoids a spatial discretization in terms of physical nodes by transforming and solving the response of the half-space in the so-called wave number domain, see further [1]. The disadvantage of this approach, however, is that the method only applies to horizontally layered ground, i.e. with horizontal interfaces and surfaces. It can therefore not be used to analyse general problems, for example inclined ground layers, an underground tunnel or the basement of a structure.

To describe wave propagation in three dimensions, the FE method can be used instead. The downside to this method being that it can become computationally expensive as the model size increases and the global system of equations must be solved simultaneously.

2.2.1 The finite element method

The FE method is a numerical method of solving partial differential equations by discretizing any given geometry into a collection of finite elements, joined by shared nodes. The collection of nodes and finite elements make up the mesh. Each finite element represents a discrete portion of the physical structure and the differential equation is solved locally for each element [3]. A detailed description of the FE method can be found in [4].

The advantages of the FE method are that it simplifies complex problems by approximating solutions using lower-order derivatives, and it accommodates certain discontinuities within elements, as long as global continuity or weak form conditions are satisfied.

The finite element formulation of a problem is derived in a number of steps, starting with the strong formulation of a problem, which consists of a differential equation along with the corresponding boundary conditions. The problem is then transformed into a weak form by introducing a weight function and integrating over the domain, with the boundary conditions incorporated implicitly and explicitly.

The approximate solution within each element is constructed using shape functions, which define how the field variables vary within an element. The shape functions are typically polynomials of a certain degree and they interpolate the solution between nodes. The unknowns in the problem are discretized at the nodes and are associated with degrees of freedom, DoFs, which represent the unknown variables, for example displacements and rotations.

The choice of shape functions directly influences the accuracy of the solution. A polynomial of degree 1 corresponds to linear elements, while a polynomial of degree 2 corresponds to quadratic elements, providing a more accurate approximation. The number of nodes in an element determines the number of terms in the polynomial approximation, which in turn influences the element's ability to capture variations in the solution.

As the element size decreases, the approximate solution converges toward the exact solution. However, when designing a FEM mesh one must balance accuracy and computational cost. This can be done by making use of symmetry in the geometry or applying knowledge of the end results by refining the mesh where significant changes in the field variables occur while keeping it coarser in regions with less variation.

2.2.2 Abaqus

Abaqus is a simulation program based on the FE method that offers a wide range of capabilities with regards to geometry, materials and problem types that can be studied for both linear and non-linear analysis. A complete Abaqus analysis consists of three stages: preprocessing, simulation and post-processing, which are linked together by files as shown in Figure 2.3. The preprocessing stage defines the model and numerical problem which is then solved in the simulation stage. The results can thereafter be evaluated in the post-processing stage [5].

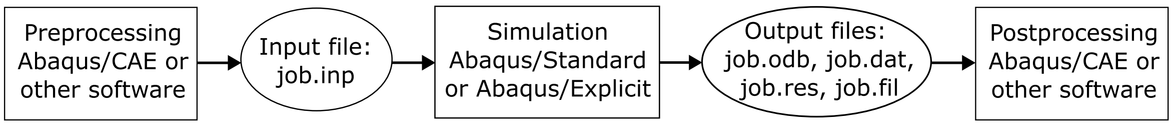


Figure 2.3: Stages and files in Abaqus.

An Abaqus model is composed of several different components that together describe the physical problem and the results to be obtained. These include a discretized geometry, element section properties, material data, loads, boundary conditions, analysis type and output requests.

Abaqus provides two different interfaces for the FE modelling. One of these is the interactive interface, which consists of either a graphical user interface or a scripting interface based on the programming language Python. The second one is the keyword interface, which is a text-based interface.

The interactive interface, Abaqus/CAE, is the Complete Abaqus Environment that allows for models to be created and meshed, as well as physical and material properties to be assigned to the geometry together with loads and boundary conditions. When the model has been generated and submitted, Abaqus/CAE generates an input file with all relevant data which can be sent for analysis [6].

The keyword interface

The keyword interface provides the same functionality as the interactive interface, but can be used independently of the Abaqus/CAE. This is achieved by the use of keywords, i.e. commands defined by Abaqus using a specific syntax in a so-called input file. An input file contains the complete description of the numerical model, where every operation in the Abaqus/CAE is converted to a keyword [7].

The input file must follow a certain structure to be valid [8]. First comes the model definition, where all the nodes and their respective coordinates are defined as well as all the elements with the nodes in each element listed in an order that aligns with the Abaqus framework for node ordering. Furthermore, all relevant element and node sets need to be defined in order to allow for material, load and boundary condition assignments. Materials need to be defined by specifying properties, followed by a definition of history data, where the step is defined, boundary conditions, loads and output requests. If multiple parts are to be used in the model, it is often useful to create instances of the parts as to avoid conflict between node numbering in the different parts.

An example of an input file for a simple ground model consisting of a cube with the sides of 1 and of only one element consisting of 20 nodes is depicted in Appendix A. Abaqus allows for a maximum of 16 values per row, which is why the remaining nodes in the element continue on a new row. The example input file specifies a steady state analysis with a load of 1 N placed on the model and the response of which is measured in the whole model as well as for a specified observation point with regards to displacements, velocity and acceleration.

The model created can be submitted for Abaqus analysis directly or be opened in Abaqus/CAE for viewing before sending the job for analysis, see Figure 2.4.

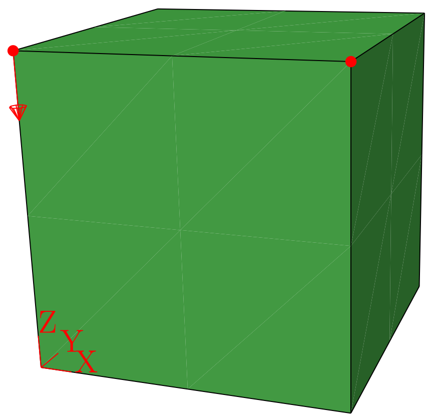


Figure 2.4: Ground model consisting of a cube with the sides of 1. Load placed in coordinates (0,0,1) and observation point in coordinates (1,0,1).

Elements

Abaqus has an extensive element library that can be used for different applications. There are five aspects that characterizes an elements behaviour. These are family, degrees of freedom, number of nodes, formulation and integration. Each element in Abaqus has a unique name that identifies each of the five aspects of an element [9].

The element family relates to the geometry type of the element and is indicated in the first letter or letters of the element's name. Relevant examples of this is the continuum elements that start with a "C", and infinite elements that start with "CIN". Degrees of freedom are, as explained in Section 2.2.1, the fundamental variables calculated during the analysis. Number of nodes in an element is significant as that is where the degrees of freedom are calculated in the element. At any other point in the element, the variables are obtained by interpolation from the nodal variables. Elements that have nodes only at their corners use linear interpolation in each direction and are therefore called linear elements or first-order elements. Elements with mid-side nodes use quadratic interpolation and are called quadratic elements or second-order elements. The number of nodes in an element is indicated in the element name. An example of a complete element name in Abaqus is C3D8, signifying a continuum, three-dimensional 8-node element.

Formulation refers to the mathematical theory used to define the element's behaviour. The standard stress/displacement element is based on the Lagrangian formulation and is for this case not reflected in the element name. All elements use numerical integration in order to allow analysis of any material behaviour. Some continuum elements can use full or reduced integration, labelled as "R" in the element name. For full integration, enough calculation points are used within each element to capture the linear material behaviour exactly. For reduced integration, fewer calculation points are used within each element. This provides accurate enough results for the main behaviour, but might miss some complex effects. Using the previous 8-node element as example, the full element name with reduced integration would be C3D8R.

Solid continuum elements

Solid continuum elements are the standard volume elements of Abaqus and may be used for linear stress analysis [10]. Relevant for this project are two types of continuum elements. The three-dimensional solid 20-node quadratic brick element with full integration, C3D20, and the axisymmetric solid 8-node biquadratic element with full integration, CAX8. An axisymmetric element is a two-dimensional element that is symmetric with respect to geometry and loading about an axis.

The node ordering in the elements is of importance as it ensures positive surfaces and volumes, see Figure 2.5.

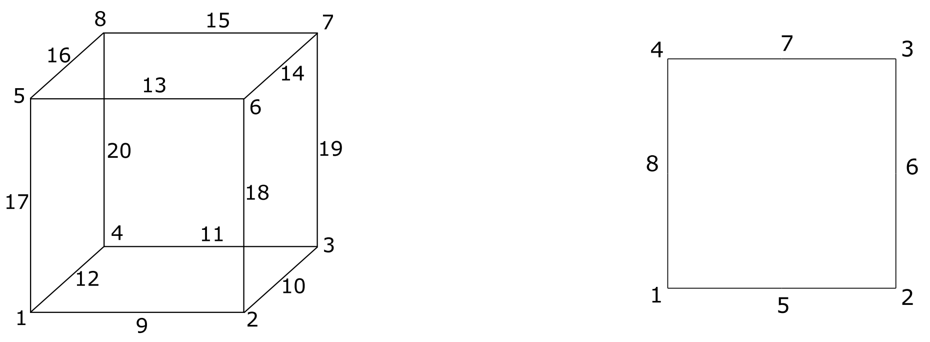


Figure 2.5: Node ordering for two continuum elements in Abaqus. Left: C3D20. Right: CAX8.

Infinite elements

When creating a FE model, boundaries are inevitable. This is a problem when wanting to model a semi-infinite domain because it leads to unwanted boundary value problems in the model. It is particularly a concern in dynamic analyses, where the boundaries will cause a reflection of energy back into the region being modelled. In Abaqus, this is solved with the use of infinite elements.

The infinite elements provide "quiet" boundaries to the finite element model in dynamic analyses by modelling the basic solution variable, in this case the displacement, with respect to spatial distance measured from a "pole" of the solution, so that the displacement moves towards zero as the distance moves towards infinity [11]. This is achieved by writing the infinite elements in Abaqus using only nodes on the interface between the finite and infinite elements and on each edge that stretches to infinity [12], see Figure 2.6.

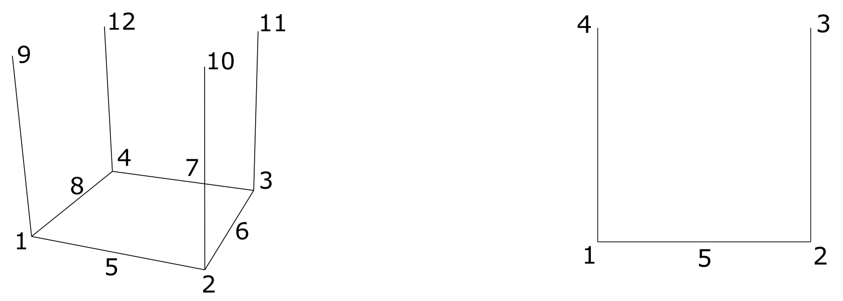


Figure 2.6: Node ordering for two infinite elements in Abaqus. Left: CIN3D12R. Right: CINAX5R.

The infinite elements are the most effective when hit by vibrations exactly perpendicular to the boundary. The sweeping direction of the elements is of importance as the elements must be stacked in the direction of the edges that stretch to infinity. Important to note, however, is that the boundaries are quiet and not silent, meaning they are not perfect transmitters and should therefore still be placed at a reasonable distance from the region of main interest [11].

2.3 Geometry- and mesh generation

Abaqus could be used for the preprocessing stage, i.e. for the model creation, as explained in the previous section. However, for the purpose of creating different sized models and elements based on the waves propagating in the medium and the frequency range that is to be studied, it would be more efficient if the proper geometry and mesh were automatically created based on the values of the influencing parameters. Not using Abaqus for the creation of the models would also allow for the models to work for all Abaqus versions and would ensure that the process is not limited by available Abaqus licenses.

As the Abaqus scripting interface is an extension of the programming language Python, the choice of Python as the tool for creating the geometry and mesh fell naturally.

2.3.1 Python

Python is an object-oriented programming, OOP, language and makes use of core OOP principles such as classes, objects, encapsulation, inheritance, polymorphism and abstraction. It also makes use of concepts such as DRY, which stands for Don't Repeat Yourself and encourages the use of functions instead of reusing the same code. These concepts are often useful when dealing with large programs as they provide a more structured code.

A class is a blueprint for creating an object. It defines a set of attributes, i.e. variables, and methods that the created object can have. An object is in turn an instance of a class. It represents a specific implementation of the class and holds its own data. The `self` parameter is used as a reference to the current instance of the class and allows one to access the attributes and methods of the object.

Encapsulation is the bundling of data and methods within a class, with the option of restricting access to some components to control interactions. Inheritance allows a class, the child class, to acquire properties and methods of another class, the parent class. This promotes code reuse and enforces the DRY principle. Polymorphism in turn allows methods in different child classes to have the same name but behave differently through method overriding, while abstraction ensures consistency in child classes by enforcing the implementation of abstract methods [13].

Python has an extensive standard library of built-in functions, found in [14], while also providing compatibility with a large amount of external libraries such as NumPy, Matplotlib, QtPy and Gmsh.

2.3.2 Gmsh

Gmsh is a three-dimensional FE geometry- and mesh generator that can be used through a number of different application programming interfaces, APIs, one of these being Python [15]. The following section will provide relevant theory on the Gmsh Python API.

Geometry

A geometry in Gmsh is defined by points, curves, surfaces and volumes, known as entities. Each entity is uniquely defined by a tuple containing two integers, the entity's dimension and its tag `(dim, tag)`. The tag is always unique per dimension. The dimensions are 0 for points, 1 for curves, 2 for surfaces and 3 for volumes [16]. The first point created would therefore be uniquely identified by the tuple `(0, 1)`.

A point is created by defining its coordinates, an optional target mesh size close to the point and an optional point tag. If a point tag is not defined, Gmsh will appoint an appropriate tag.

```
gmsh.model.geo.addPoint(x-coord, y-coor, z-coord, target mesh size, point
↪  tag)
```

Curves, and in this case specifically straight lines, are created by specifying the two point tags that will be the start and end points of the line along with an optional line tag. The line tags are separate from point tags and the same numbers can therefore be reused.

```
gmsh.model.geo.addLine(point tag of start point, point tag of end point,
↪  line tag)
```

For the third elementary entity, the surface, a curve loop must first be defined by an ordered list of connected curves. The curves must be listed with correct signs that ensure counter-clockwise orientation of the surface. The plane surface can thereafter be defined by the use of the curve loop tag.

```
gmsh.model.geo.addCurveLoop([line tag1, line tag2, line tag3, line tag4],
↪  curve loop tag)

gmsh.model.geo.addPlaneSurface([curve loop tag], surface tag)
```

Lastly, a volume can be created by, in the same manner as for surfaces, first defining a surface loop containing a list of surfaces that enclose the volume and then creating the volume by the use of the surface loop tag.

```
gmsh.model.geo.addSurfaceLoop([surface1, surface2, surface3, surface4,
↪  surface5, surface6], surface loop tag)

gmsh.model.geo.addPlaneSurface([surface loop tag], volume tag)
```

Gmsh offers different approaches to creating entities in the geometry. One of these is manually creating every point, line, surface and volume in the geometry in a manner as explained above. This approach offers complete control of the created entities and tags, it is however rather time consuming. For a simple cube, this method would require the creation of 8 points, 12 lines, 6 curve loops/surfaces, and 1 surface loop/volume.

An alternative method for the creation of geometries in Gmsh is extruding entities to create entities of higher dimensions. A point could be extruded to a line, a line to a surface, and a surface to a volume. This is done using the Gmsh extrude function with arguments for the entity to be extruded, the translations in the x- y- and z- direction for the extrusion, and the number of elements to be created in the extrude direction.

```
gmsh.model.geo.extrude([entity(dim, tag)], dx, dy, dz, numElements=[])
```

This method could be used to create a cube with the sides 1.

```
gmsh.initialize()

#Create a point
gmsh.model.geo.addPoint(0, 0, 0, 0, 1)

#Extrude point to a line
line = gmsh.model.geo.extrude([(0,1)], 1, 0, 0, numElements=[1])

#Extrude line to a surface
surface = gmsh.model.geo.extrude([line[1]], 0, 1, 0, numElements=[1])

#Extrude surface to a volume
volume = gmsh.model.geo.extrude([surface[1]], 0, 0, 1, numElements=[1])

gmsh.model.geo.synchronize()
```

The extrude function returns a list of tuples `(dim, tag)` with the created entities from the extrusion. The extrusion in the example will return:

```
line = [point, line]
surface = [1st line, surface, 2nd line, 3d line]
volume = [top surface, volume, 1st side surface, 2nd side surface, 3d side
↪  surface, 4th side surface]
```

The extrude function creates the same geometry with less code, with the downside of possessing less control of the geometry. However, the methodologies could also be combined, using the first method where more control is needed and the second method otherwise.

Another relevant Gmsh feature is physical groups, which allows for groups of entities to be defined. This is particularly useful when wanting to define certain parts of the geometry with regards to, for example, material or element type. Physical groups are, in the same manner as entities, defined by a tuple containing the physical group's dimension and tag.

```
gmsh.model.addPhysicalGroup(dim, [entity tags], physical group tag, name="")
```

The entities for the model can be extracted as a list of tuples using `gmsh.model.getEntities()`, after which the physical group of an entity can then be checked using `gmsh.model.getPhysicalGroupsForEntity(entity dim, entity tag)`.

Elements and mesh

By default, meshes produced by Gmsh are unstructured and the elements generated are linear triangles for surfaces and linear tetrahedra for volumes. In order to guarantee conformity in the mesh, the mesh generation is performed in a bottom-up flow, where curves are discretized first and the mesh of the curves is then used to mesh the surface, which in turn is used to mesh the volumes.

A structured mesh can be achieved using the extrusion function previously mentioned, and/or by using transfinite commands while creating the entities.

```
gmsh.model.geo.mesh.setTransfiniteCurve(line tag, nbr of points on the line)

gmsh.model.geo.mesh.setTransfiniteSurface(surface tag)

gmsh.model.geo.mesh.setTransfiniteVolume(volume tag, [corner point tags])
```

In order to obtain quadrangles instead of triangles, the recombination command can be used. The element order could then be set to give incomplete second order quadrangle elements, i.e. a 20-node brick or 8-node quadrangle [16], see Figure 2.7.

```
#recombine all triangles into quads
gmsh.option.setNumber('Mesh.RecombineAll', 1)

#set second order elements
gmsh.option.setNumber('Mesh.ElementOrder', 2)

#set incomplete second order elements
gmsh.option.setNumber('Mesh.SecondOrderIncomplete', 1)
```

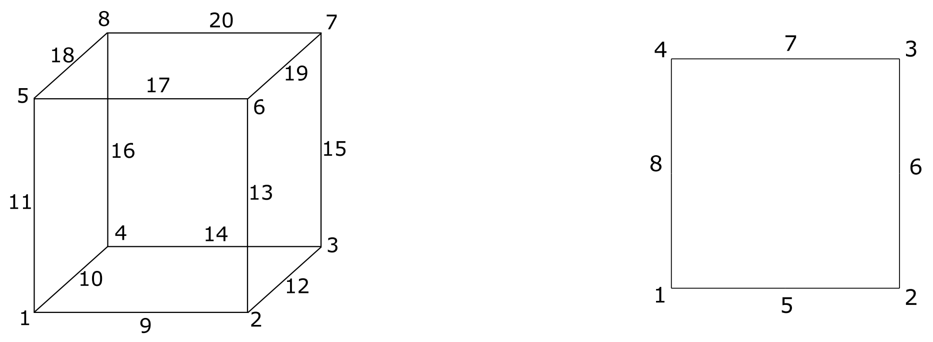


Figure 2.7: Node ordering for two second order quadrangle elements in Gmsh

Once the mesh is generated, mesh data related to the nodes and elements can be extracted either for all entities, or for a specific entity by using the entity's dimension and tag as arguments to the functions.

```
nodeTags, nodeCoords, nodeParam = gmsh.model.mesh.getNodes(dim, tag)
```

```
elemTypes, elemTags, elemNodeTags = gmsh.model.mesh.getElements(dim, tag)
```

3 Program structure

The program for the analysis of ground vibrations in Abaqus was developed using Python.

The program starts with `class BaseModel`, which serves as a blueprint for two models, the axisymmetric and the three-dimensional (3D) model. This class defines common parameters and includes methods to calculate model- and element sizes based on wavelengths. Two specialized model classes inherit from this base class, `class Axisymmetric` and `class Model3D`. Each specialized class implements the specific geometry creation and mesh generation for its model type. Once the model is created, `class MeshData` extracts all relevant information from the mesh. `class Visualization` allows for visual inspection of the created models.

`class InputFile` uses the model- and mesh data to generate Abaqus input files. For more complicated analyses across frequency ranges, `class TailoredInputFiles` creates multiple input files by dividing the frequency range into bands and updating the model for each band.

`class RunAbaqus` handles job submission to Abaqus, either in series or parallel, and monitors job status. After Abaqus completes the analyses, an external script extracts data from the output database files using `class ODBData`. Finally, `class Results` processes this data and plots the results. Figure 3.1 depicts the work flow and class relationships.

The program can be managed from a user interface, created to improve user-friendliness.

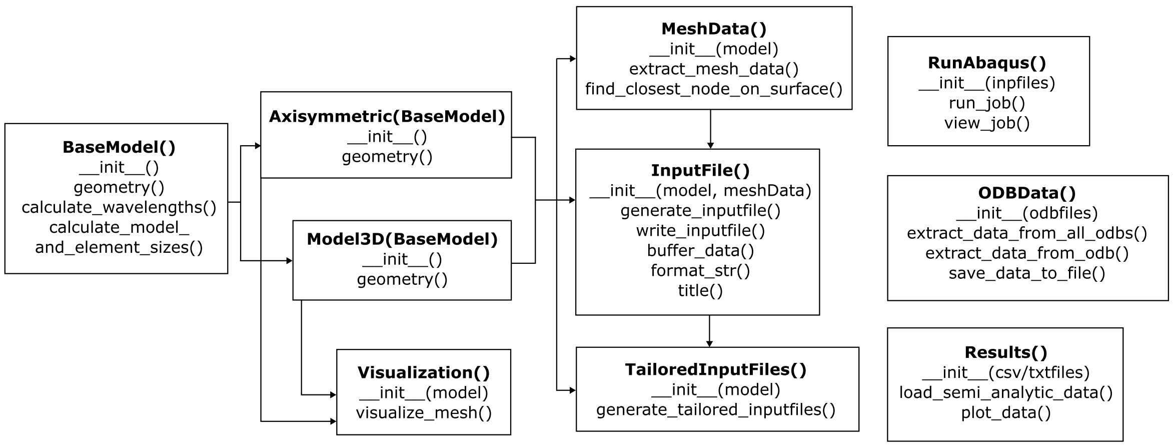


Figure 3.1: Work flow diagram and class relationships.

3.1 Creating the models

The creation of the models starts with the parent class `BaseModel`. The constructor of this class defines all of the relevant common parameters for the models, divided into parameters defined by the user and parameters that are calculated by the class.

The user-defined parameters encompass properties of three ground layers. These properties include depth, inclination, density, Young's modulus, Poisson's ratio, and loss factor. They also include frequency-related specifications such as the upper and lower bounds of the frequency interval, size of frequency increments and size of frequency bands. The size of the ground domain is determined by parameters defining the number of P-wavelengths in each direction, the distance between the loading and observation point, and the length and width of an optional building. Element size is determined by the number of elements per Rayleigh wavelength. Finally, the size of the load is defined.

The stored values that are calculated by the class include the size of the model in all directions, the height difference in each layer's surface due to an inclination, the element size and the depth of the infinite elements.

```
class BaseModel():
    def __init__(self):
        self.dim = None
        self.nbr_of_nodes_per_element = None

        '''User defined parameters'''
        self.depth = [4, 5, 5]
        self.inclination = [0, 0, 0, 0]
        self.rho = [2000, 2000, 2000]
        self.E = [160e6, 800e6, 800e6]
        self.v = [0.33, 0.33, 0.33]
        self.eta = [0.06, 0.06, 0.06]

        self.lower_freq = 50
        self.upper_freq = 55
        self.freq_increment = 0.5
        self.freq_band = 5

        self.nbr_of_wavelengths_in_model = 1
        self.nbr_of_elements_per_wavelength = 5
        self.dist_load_op = 10
        self.building_length = 0
        self.building_width = 0
        self.force = None
```

```
'''Calculated values'''
self.x = None
self.x_each_dir = None
self.y = None
self.z = []
self.element_size = None
self.tot_depth = None
self.inf = None
```

The `BaseModel` class contains three methods, one for calculating wavelengths, one for calculating model- and element sizes and one initializing the creation of the geometry.

`def calculate_wavelengths()` uses the material properties defined in the constructor to calculate and store the relevant wavelengths for each of the three layers in lists. The procedure starts with calculating the wave speeds in each of the three layers. For the P-wave and S-wave this is done using Equations 2.1 and 2.2. The wave speed for the Rayleigh wave is approximated using the design case of Poisson's ratio being $\nu = 0$, which translates into $c_R \approx 0.862c_S$. The P-wavelengths and Rayleigh wavelengths for each layer are calculated using Equation 2.3 and are then stored in their respective list. Finally, the function returns the maximum P-wavelength and minimum Rayleigh wavelength that occurs in any of the three layers.

`def calculate_model_and_element_sizes()` invokes the previous function and uses its return values to calculate the model- and element sizes.

The calculations of the model size differs depending on the model type. For the axisymmetric model, the x-axis has its origin at the loading point, and the size of the model in the x-direction is calculated as the sum of the distance between the loading and observation point and the maximum P-wavelength multiplied by the user-specified number of wavelengths, see Figure 3.2.

```
if self.dim == 2:
    self.x = max_p * self.nbr_of_wavelengths_in_model + self.dist_load_op
```

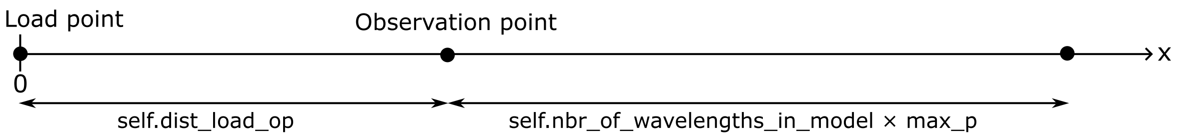


Figure 3.2: The x-axis of the axisymmetric model.

In order to provide the possibility of including a building to the 3D model, the x-axis is constructed a bit differently. The origin starts in the observation point. In the positive x-direction, the model extends to the sum of half of an optional building's length and the maximum P-wavelength multiplied by the user-specified number of wavelengths.

In the negative x-direction, the model extends to the sum of the distance between loading and observation point and the maximum P-wavelength multiplied by the user-specified number of wavelengths, thus ensuring the same distance from the boundaries to the loading point and the outer edge of a possible building respectively, see Figure 3.3. The 3D model is single-symmetric about the x-z plane and the size of the model in the y-direction is therefore only calculated in one direction as the sum of half the building's width and the maximum P-wavelength multiplied by the user-specified number of wavelengths.

```
if self.dim == 3:
    self.x_each_dir = max_p * self.nbr_of_wavelengths_in_model
    self.x = 2 * self.x_each_dir + self.dist_load_op +
    ↪  (self.building_length/2)
    self.y = max_p * self.nbr_of_wavelengths_in_model + self.building_width
```

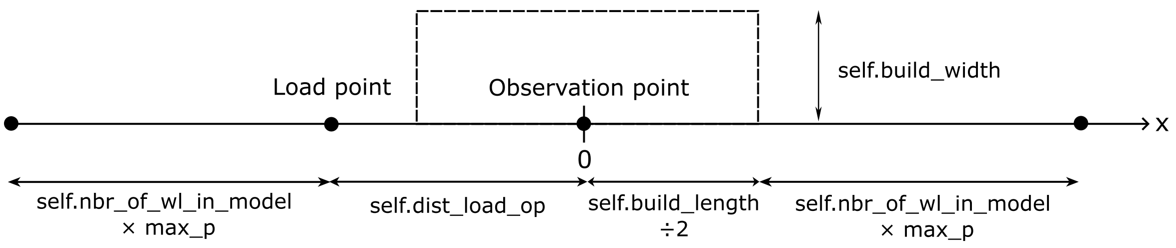


Figure 3.3: The x-axis of the 3D model.

The rest of the calculations are the same for both model types. The depth of the third layer is adjusted to align the total depth with the calculated value for the model size in each direction. The element size is decided as the length of the minimum Rayleigh wavelength divided by the user-specified number of elements per wavelength. The height difference on a layer's surface due to an inclination is calculated as the x-distance multiplied by the tangent of the angle. The depth of the infinite elements is set as a tenth of the total depth of the model.

```
self.tot_depth = self.nbr_of_wavelengths_in_model*max_p
#Adjusting the depth of layer 3
self.depth[2] = max(0.1, self.tot_depth-(self.depth[0]+self.depth[1]))
#Depth of infinite elements
self.inf = self.tot_depth / 10
#Mesh size dependent on Rayleigh wavelength
self.element_size = min_r / self.nbr_of_elements_per_wavelength
#Height difference in layer boundary due to inclination
for inc in self.inclination:
    inc_in_rad = math.radians(inc)
    self.z.append(self.x * math.tan(inc_in_rad))
```

Finally, `def geometry()` invokes the function `calculate_model_and_element_sizes()`, sets up the initialization of Gmsh and arranges so that the mesh created by child classes will contain incomplete second order elements, i.e. an 8-node quadrangle for the axisymmetric model or a 20-node brick for the 3D model, according to Section 2.3.2.

3.1.1 Axisymmetric model

The `Axisymmetric` class inherits from the `BaseModel` class and reuses the attributes defined in this parent class by calling its constructor through `super().__init__()`, which is known as constructor chaining. The class overrides some of the attributes in order to make them model specific, such as the dimension, number of nodes per element and the force.

```
def __init__(self):
    super().__init__()
    self.dim = 2
    self.nbr_of_nodes_per_element = 8
    self.force = 1
```

The class also extends the parent class's implementation of the `geometry()` method by calling `super().geometry()` before adding its own functionality. In order to possess more control over the created geometry, and thereby simplifying the possibility of inclined ground layers or more complicated geometries, the manual approach as described in Section 2.3.2 is applied to the parts of the model that will contain finite elements.

First, points are defined by coordinates. For points along the right boundary of the model, where the x-coordinate is equal to the model width, see Figure 3.4, the y-coordinates are determined by the inclination of each layer. A positive inclination value results in an elevated y-coordinate, while a negative inclination produces a lower y-coordinate, relative to a horizontal orientation. This relationship is quantified by the tangent of the inclination angle multiplied by the x-distance, as previously calculated in the `BaseModel` class. The points along the left boundary, at $x = 0$, maintain consistent depths corresponding to the cumulative thickness of the overlying layers.

Between the points, horizontal and vertical lines are defined and set as transfinite in order to create a structured mesh. For the horizontal lines, i.e. the lines running in the x-direction, the number of points are set to `x/element_size` rounded upwards to an integer. For the vertical lines, the number of points are calculated using the maximum depth of each layer, taking a possible inclination into consideration, divided by the element size and rounded upwards to an integer.

```
#Vertical lines
gmsh.model.geo.mesh.setTransfiniteCurve(line,
↪   math.ceil(max_depth_in_layer/element_size + 1))
```

The curve loops are defined in a counter-clockwise manner and plane surfaces are thereafter created and set as transfinite.

As less control is needed for the infinite domains, these parts of the model are created using the extrude method. The lines that make up the boundaries that should be modelled as infinite are simply used as a base in the extrude function with the number of elements in the extrude direction set to 1, see Figure 3.4. The direction of the extrusion for the infinite domain is important, as this will affect which side of the elements that go towards infinity, as explained in Section 2.2.2. Extruding perpendicular to the side that should go towards infinity, as done in the procedure described above, will ensure the right direction.

For future analytical purposes, a categorical distinction is established between geometric entities, in this case surfaces, containing finite elements versus those containing infinite elements. The surfaces are also divided according to their respective ground layer. To enable this classification system, data structures are initialized prior to geometry creation, a list for finite surfaces, another for infinite surfaces and a nested list comprising of three distinct lists corresponding to each ground layer. During the geometric construction process, each surface entity is then systematically categorized and stored in both the appropriate element type collection and its corresponding layer within the nested list. These categorized surfaces are then used to create six physical groups of entities: ’Finite ground’, ’Infinite ground’, ’Layer 1’, ’Layer 2’, ’Layer 3’ and ’Ground surface’, referring to the upper most line in the model constituting the ground surface.

Finally, the mesh is created using the Gmsh generate function with the model dimension as an argument, `gmsh.model.mesh.generate(2)`. The model can be visualized using the `Visualization` class, which takes an instance of the model as an input parameter, creates the geometry for the instance and uses a `Gmsh` command to visualize the model, Figure 3.4.

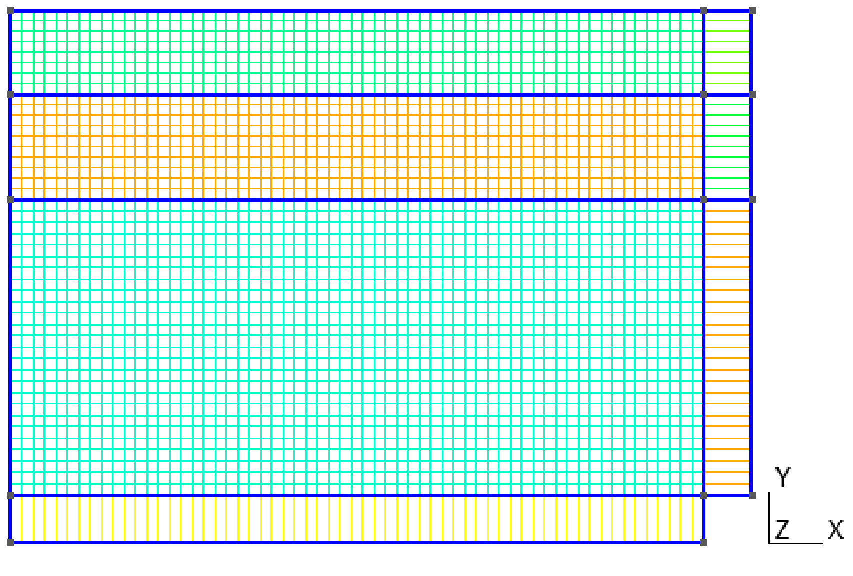


Figure 3.4: The axisymmetric model, symmetric about the y-axis and consisting of three ground layers with infinite elements to the right and bottom.

3.1.2 3D model

The creation of the 3D model follows the same procedure as described for the axisymmetric model, albeit with further complexion. Starting with the constructor, the same method of constructor chaining with the parent class `BaseModel` applies. However, to achieve a load equivalent to the axisymmetric model, the size of the load is halved due to the 3D model being single-symmetric and the load being placed in the symmetry plane.

```
def __init__(self):
    super().__init__()
    self.dim = 3
    self.nbr_of_nodes_per_element = 20
    self.force = 0.5
```

The number of points, lines and surfaces are significantly increased for this model and are dealt with by an increased use of for-loops. As in the case of the axisymmetric model, the z-coordinates of the furthest right points in the finite domain are dependent on the inclination. The transfinite lines in the y-direction, see Figure 3.5 are, equivalently to the x-direction, set to the rounded up integer of `y/element_size`.

```
gmsh.model.geo.mesh.setTransfiniteCurve(line, math.ceil(y/element_size + 1))
```

Furthermore, volumes are created for each layer by the use of surface loops and setting transfinite volumes using the corner points for the respective layer.

The infinite domains are extruded from four different directions of the model, in order to ensure the correct sweep direction. This calls for the need to keep track of all side surfaces created in the model, which is done by storing the surfaces when creating them in a nested list consisting of three lists, one for each layer. The values in the list are then used for extruding the infinite domains with 1 element in the extrude direction.

```
#[[layer1(right, back, left)], [layer2], [layer3]]
infinite_base_surfaces = [[], [], []]

x_extr = [inf, 0, -inf]
y_extr = [0, inf, 0]

for layer in range(3):
    for side in range(3): #right, back, left
        extr_vol = gmsh.model.geo.extrude([(2,
         ↪  infinite_base_surfaces[layer][side])], x_extr[side],
         ↪  y_extr[side], 0, numElements=[1], recombine=True)
```

In the same manner as explained for the axisymmetric model, the volumes for the finite domain, infinite domain and each ground layer are stored. Additionally, the surfaces coinciding with the symmetry plane, i.e. the x-z plane, are also stored. Physical groups are created for the finite domain, the infinite domain, layers 1-3, the ground surface, and the symmetry plane respectively. Lastly, the mesh is generated using `gmsh.model.mesh.generate(3)`, see Figure 3.5.

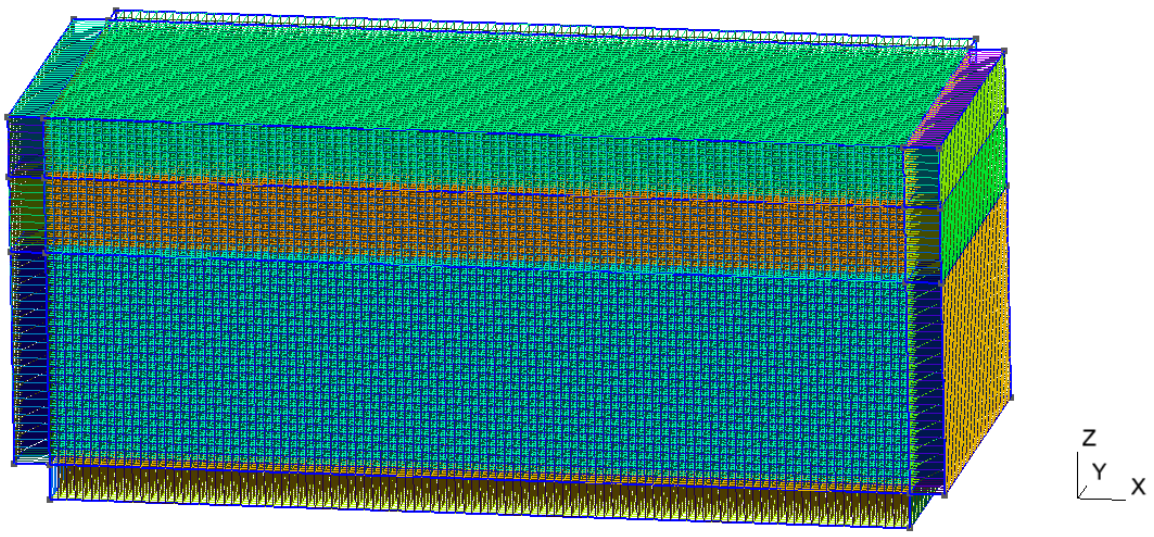


Figure 3.5: The 3D model, single-symmetric about the x-z plane with infinite elements applied in all other directions.

3.2 Extracting mesh data

The `MeshData` class takes an instance of either the axisymmetric or the 3D model as an input parameter. Its constructor contains attributes for all the data that needs to be extracted from the mesh of the model for the input file to be valid, see Section 2.2.2 and Appendix A.

For both model types, these attributes include all the nodes in the model with their respective coordinates, the finite elements with their respective node tags ordered according to the Abaqus framework, the infinite elements with correct node ordering, the elements in layers 1-3 respectively, the nodes that make up the ground surface of the model and their respective coordinates, the node tag used as the loading point and the node tag used as the observation point. Additional attributes are needed for the 3D model, one listing all the node tags on the surface, i.e. only the tags and not the coordinates, and one listing all the node tags on the symmetry plane.

The attributes that are to be assigned as either NumPy arrays or a single value are given the initial value `None` while the rest of the attributes are created as empty lists. The attributes are updated after mesh data has been extracted through the `extract_mesh_data()` function.

```python
class MeshData():
    def __init__(self, model: object):
        self.model = model
        self.model.geometry()

        '''Attributes relevant for both models'''
        self.all_nodes = None
        self.fin_elements = None
        self.inf_elements = None
        self.el_layer1 = []
        self.el_layer2 = []
        self.el_layer3 = []
        self.surface_nodes = None
        self.loading_point = None
        self.observation_point = None

        '''3D model specific attributes'''
        self.surface_node_tags = []
        self.sym_plane = []

        self.extract_mesh_data()
```

The `extract_mesh_data()` method extracts data from the mesh using the functions explained in Section 2.3.2. The attributes `self.all_nodes`, `self.fin_elements`, `self.inf_elements` and `self.surface_nodes` are created as NumPy arrays, due to the data for these needing to be arranged in specific rows and columns, which can be achieved using the NumPy functions `reshape()`, `hstack()` and `vstack()`. The other attributes containing multiple data do not demand a specific order and are for simplicity created as lists.

Categorized data is achieved by looping through every entity in the model, i.e. every point, line, surface and volume, and identifying which physical groups it belongs to, followed by extracting the elements and nodes in the entity and storing them in the appropriate data structures.

```python
entities = gmsh.model.getEntities()
for entity in entities:
    dim, entity_tag = entity
    physical_group_tags = gmsh.model.getPhysicalGroupsForEntity(dim,
    ↪   entity_tag)

    _, elemTags, elemNodeTags = gmsh.model.mesh.getElements(dim, entity_tag)
    node_tags, node_coords, _ = gmsh.model.mesh.getNodes(dim, entity_tag,
    ↪   includeBoundary=True)
```

For the 3D model, the node tags in the finite and infinite element arrays need to be reordered as there is a discrepancy between node ordering in Gmsh and Abaqus, see Figure 3.6. The excessive nodes in the infinite elements must also be deleted. The node ordering for the 8-node quadrangle is the same for both software tools and therefore do not need to be rearranged.

```
if self.model.dim == 3:
    self.fin_elements = self.fin_elements[:, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    ↪  12, 14, 10, 17, 19, 20, 18, 11, 13, 15, 16]]
    self.inf_elements = self.inf_elements[:, [0, 1, 2, 3, 4, 9, 12, 14, 10,
    ↪  5, 6, 7, 8, 17, 19, 20, 18, 11, 13, 15, 16]]
    self.inf_elements = np.delete(self.inf_elements, slice(13,21), axis=1)
```

The first column in the arrays, `[0]`, contains the element tags. The node tags start from the second column, `[1]`, and correlate with the node tags in Figure 3.6.

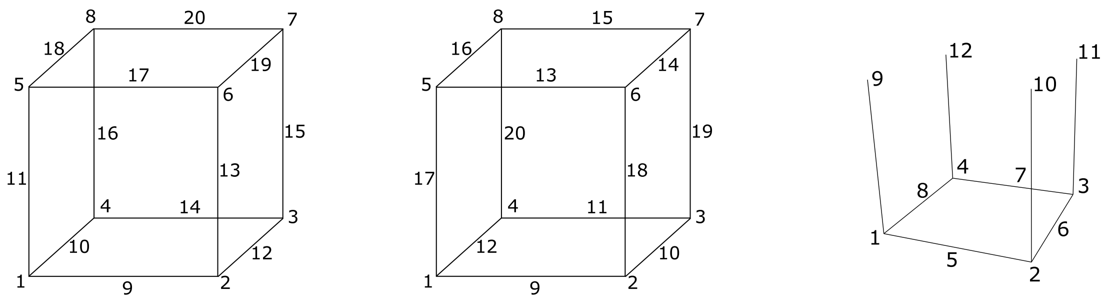


Figure 3.6: Comparison of node ordering in Gmsh, left element, and Abaqus, two right elements.

The nodes used as the loading and observation points, respectively, are determined by the use of the `find_closest_node_on_surface(x_value)` method, which takes a x-coordinate as an argument and returns the node on the x-axis that is located closest to this value.

The node used as loading point for the axisymmetric model is set as the node closest to the point $(0, 0)$. A point was defined with these coordinates in the creation of the axisymmetric model, the loading point in this model will therefore always have these exact coordinates. The node used as the observation point will however be approximate to the user-specified x-coordinate, with the `find_closest_node_on_surface()` function finding the node on the x-axis closest to the x-value that corresponds to a given distance between the loading and observation point defined by the user.

In the 3D model, the observation point is chosen as the node on the x-axis closest to the coordinates $(0, 0, 0)$. This is to ensure that a building can be added to the model correctly. The loading point is set as the node on the x-axis closest to the x-coordinate relating to a given distance between the loading and observation point.

The user-specified distance between loading and observation point will therefore be approximate for both model types. However, the exact distance typically only deviates from the user-specified distance by a few centimetres and is apparent from the created input file, see Section 3.3.1.

3.3 The input file

3.3.1 Writing the input file

The `InputFile` class takes a model instance and a `MeshData` instance as input parameters and uses these as attributes, as well as a `Boolean` attribute for including a building.

```
class InputFile():
    def __init__(self, model: object, meshData: object):
        self.model = model
        self.mesh = meshData
        self.incl_building = False
```

The class contains methods to format data, write an input file and generate an input file with an optional file name.

`def write_inputfile()` starts off by opening a file and starting writing to it with the use of help functions, attributes from the model instance and the data structures from the `MeshData` instance. In the beginning of the input file, useful information about the model and the mesh is written out, see example below.

```
**--------------------------------------------------------------------------------
** Number of nodes: 1011356
** Number of elements: 243354
** Number of wave lengths in each direction: 2.0
** Number of elements per wavelength: 4
** Model size: x = 71.59 m, y = 30.79 m, depth = [4.00, 5.00, 21.79] m
** Inclination: [0.0, 0.0, 0.0, 0] degrees
** Mesh size: 0.68 m
** E = [1.60e+08, 8.00e+08, 8.00e+08] Pa
** Density = [2000.0, 2000.0, 2000.0] kg/m3
** Poissons = [0.33, 0.33, 0.33]
** Structural damping = [0.06, 0.06, 0.06]
** Frequency step: 0.5 Hz
** Horizontal distance between load and observation point: 10.13 m
** Height difference between load and observation point: 0.00 m
** Force: 0.5 N
**--------------------------------------------------------------------------------
```

The file is then written following the structure in Section 2.2.2 and Appendix A. Abaqus limits the number of numerical values on each line to a maximum of 16, for rows of data containing more values than this the data is formatted in the `buffer_data` function. Data relating to element definition need to end the first line of data with a comma and then continue the data on the line underneath, letting Abaqus know that the element definition is not finished. See the example below of an element definition, where "239" is the element tag and the following 20 values are the node tags.

```
*Element, type=C3D20, elset=Set-fin_el
239, 815, 45, 1, 122, 3207, 1795, 381, 1942, 890, 60, 128, 891, 3283, 1810, 1958,
     3284, 3282, 1811, 382, 1956
```

The data structures that only serve as element- or nodal sets and do not define a part of the mesh is listed in rows of 16 values with no comma after the end value, see the example below.

```
*Elset, elset=Set-layer1
239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254
255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270
```

As the name indicates, the `buffer_data()` function is also used to efficiently write data to the file by the use of a buffer, i.e. a temporary storage, of 1000 in order to minimize the number of write operations to file. Initially, the data was organized into tables using the library Tabulate. However, this proved to be significantly less efficient and was discarded considering its more of a custom rather than a requirement in Abaqus to list values in right aligned columns. All text in the file was also initially built as a single string in the program's memory before writing it over to the file. As one input file can contain million of text lines, this was discarded in favour of writing each line directly to file. With these changes the generation of an input file was up to 80 times more efficient, taking 10 seconds instead of 13 minutes.

The `InputFile` class offers the option of writing an input file with a building included. This is determined by the Boolean attribute `self.incl_building`, which if set to `True` will write additional lines to the input file making it compatible with the input file of a building. The additional lines added are ones for creating parts for the building, including the input file of the building using the Abaqus `*include` keyword, adding a surface set for the nodes on the surface of the ground model, a surface set for the nodes in the building footings and a tie constraint between these two using the building footing set as a slave surface and the ground surface set as the master surface. Additional material definitions are also added and applied to the building parts and symmetry boundary conditions are set for the building. Lastly, the observation point is changed to a point defined in the building's input file.

For this option to be viable, the building must have the same coordinate system as the ground model, be single-symmetric about the x-z plane and have its observation point in the x- and y- coordinates 0. Its input file must contain only instances and sets appointed to the instances. Furthermore, the building's input file must be placed in the same folder as the input file for the created ground model when running the analysis in Abaqus.

3.3.2 Generating input files

The `TailoredInputFiles` class is responsible for the creation of multiple input files in a frequency range, adjusting the model- and element sizes for each created input file. The class takes a model instance as an input parameter and uses it as an attribute along with two `Boolean` variables.

```
class TailoredInputFiles():
    def __init__(self, model):
        self.model = model
        self.scale_layer2 = True
        self.incl_building = False
```

The class contains only one method, `def generate_tailored_inputfiles(folder_path)`, which takes a directory path as an input parameter and uses it as a location for where to save the generated files. The method creates a number of frequency bands in the given frequency interval and loops through each band. For each frequency band, a new model instance is created that recalculates the model- and element sizes using the methods in the `BaseModel` class. The new model instance is used as an input parameter for a new `MeshData` instance, which in turn is used together with the model instance to create a new `InputFile` instance. An input file is written and generated using the methods in the `InputFile` class and the loop restarts using the next frequency band.

The `Boolean` attribute `self.scale_layer2` determines how the depth should be dealt with when the model size decreases for frequency bands located in higher frequency intervals. For each decrease in size, the depth of the third layer will be resized, but when the model size has decreased to the point of the depths of the first and second layers exceeding the total depth correlating to the chosen number of wavelengths, `self.scale_layer2` determines whether the second layer should be resized as well. If set to `True`, the depth of the second layer is decreased in order for the total depth to be accurate, if set to `False` the second layer will not be resized and the total depth will be larger than the chosen number of wavelengths. The minimum possible depth for the second and third layer respectively when resized is 0.1 m.

```
if self.scale_layer2 == True:
    if current_model.depth[0] > current_model.tot_depth:
    #if layer 1 is deeper than total depth should be
        current_model.depth[1] = 0.1
    else:
        current_model.depth[1] = max((current_model.tot_depth -
        ↪  current_model.depth[0]), 0.1)
    current_model.depth[2] = max((current_model.tot_depth -
    ↪  current_model.depth[0] - current_model.depth[1]), 0.1)

elif self.scale_layer2 == False:
    current_model.depth[1] = self.model.depth[1]
    current_model.depth[2] = 0.1
```

`self.incl_building` determines which `Boolean` value the attribute with the same name in the `InputFile` instance should be set to when creating the input file, determining if a building is to be included or not.

3.3.3 Submitting input files for analysis

The input files are submitted for analysis through the `RunAbaqus` class, which takes a list of either one or multiple input files as an input parameter and uses these as attributes. The constructor also contains lists for the names and paths of the input files as well as a variable for the elapsed time for an analysis and variables handling parallel jobs.

```
class RunAbaqus():
    def __init__(self, files):
        self.files_basename = [os.path.basename(file) for file in files]
        self.files_directory = [os.path.dirname(file) for file in files]
        self.total_elapsed_time = None
        self.run_parallel = False
        self.max_parallel_jobs = 4
```

`def run_job()` provides the functionality of submitting jobs to Abaqus. If `self.run_parallel` is set to `False`, the jobs will be submitted to Abaqus sequentially using the module `subprocess`, which starts and handles external processes, and the command `subprocess.run()`. Each job is run using 4 CPUs and is submitted using its input file name without the .inp suffix in the specified directory.

```
for i, job_name in enumerate(self.files_basename):
    directory = self.files_directory[i] or '.'

    job_name_base = job_name.replace(".inp", "") if
    ↪  job_name.endswith(".inp") else job_name

    subprocess.run(
        f'abaqus job={job_name_base} cpus=4 interactive', shell=True,
        ↪  cwd=directory)
```

If `self.run_parallel` is set to `True`, the jobs will run in parallel, however only four at a time as to not overload the computer. In this case, the command `subprocess.Popen()` is used instead, offering more control and allowing parallel processes. Each job is run using only one CPU.

```
subprocess.Popen(f'abaqus job={job_name_base}', shell=True, cwd=directory)
```

At regular intervals, the processes are checked to see if they have been completed by controlling if a sta-file has been created and if so, reading it to control whether the text "COMPLETED" is present in the file, marking it as completed and submitting a new job in its place.

For both sequential and parallel processes, the time it takes for all jobs to be completed is measured and stored in `self.total_elapsed_time`.

`def view_job()` allows for the finished job to be viewed in Abaqus/CAE using the created odb-file for the job.

```
os.system(f'abaqus viewer database={job_name}')
```

3.4 Post-processing of results

When a simulation is run in Abaqus, the results are stored in a output database file, i.e the odb-file, see Figure 2.3. The odb-file can be used to post-process the results in Abaqus/CAE using the `view_job()` function in the `RunAbaqus` class. Alternatively, the results can be processed by extracting the data from the odb-files and plotting them using Python.

3.4.1 Extracting results from odb-files

The extraction of data from odb-files is performed in a separate Python script. This is due to the Abaqus Python library not being compatible with a lot of the libraries and syntax used in the program, for example the QtPy library used for creating the user interface, see Section 3.5, and f-strings used continuously throughout the main script. The separate script contains the class `ODBData()` which takes a list of odb-files as an input parameter and creates an array of frequency data and a dictionary containing different types of result data.

```
class ODBData():
    def __init__(self, odb_files):
        self.odb_files = odb_files
        self.freq_array = None
        self.data_dict = {}
```

The `extract_data_from_odb()` function opens an odb-file and extracts the steps and regions from which output was requested. The steps and regions in the odb-files are contained in dictionaries. As the input file contains a single step, `Step-1`, the dictionary for the steps in the odb-file contains only one key. This key is extracted and saved in a parameter, which in turn is used to extract the key for the region where history output was requested, i.e. the observation point.

```
odb = openOdb(path=odb_path, readOnly=True)
step_name = odb.steps.keys()[0]                 #only 1 step
step = odb.steps[step_name]
region_key = step.historyRegions.keys()[0]      #only 1 region
region = step.historyRegions[region_key]
```

The output in the region, i.e in the node, is then looped through and the real and imaginary data for displacements, velocities and accelerations is extracted and converted to magnitudes using Equation 2.8.

The magnitudes and corresponding frequencies are stored as values in `self.data_dict` while the variable names, for example A1, are stored as keys.

The possibility of extracting data from multiple odb-files is given by the `extract_data_from_all_odbs()` function, which loops over a list of odb-files and invokes the `extract_data_from_odb()` function for each file, adding data to the `self.data_dict` dictionary.

The `save_data_to_file()` function takes a file name as an input parameter and writes the `self.data_dict` dictionary to a csv-file with a given file name.

3.4.2 Formatting and plotting the results

Going back to the main script, the results are formatted and plotted in the `Results` class, which takes a list of csv- or txt-files as an input parameter.

```
class Results():
    def __init__(self, files):
        self.files = files
        self.semi_analytical = False
        self.incl_building = False
        self.freq_lower = None
        self.freq_upper = None
        self.data_list = []
```

The class contains three methods, one for loading and formatting the data from the chosen files, one for loading and formatting data from the semi-analytical Thompson-Haskell model, see Section 2.2, and one for plotting the desired data.

The data from the csv-files are formatted in `load_data_fem_models()`. When data from multiple odb-files are written to a csv-file, the file will contain double sets of data for the frequencies located at the bounds between two neighbouring frequency bands. For example, a csv-file created from two odb-files containing the frequency intervals 50-55 Hz and 55-60 Hz respectively, will have a double set of data for the frequency 55 Hz when written over to the csv-file. This is dealt with in the function by using the mean value of the data with double sets. The upper and lower frequencies in the data from the file are stored as attributes and the data is stored in `self.data_list`.

If the attribute `self.semi_analytical` is set to `True`, the `load_semi_analytic_data()` function loads the data from the semi-analytical model, either with a building included or without, as determined by the attribute `self.incl_building`, and filters which frequencies to be used based on the frequency interval used in the FE model. The semi-analytical data can thereby be used for comparison and validation of the created models.

The chosen data is plotted in `plot_data()` using the Matplotlib library. In the 3D model, the data for the y-direction, i.e. the horizontal direction perpendicular to the symmetry plane, is not plotted due to the observation point being placed in the symmetry plane and all results in this direction therefore being zero.

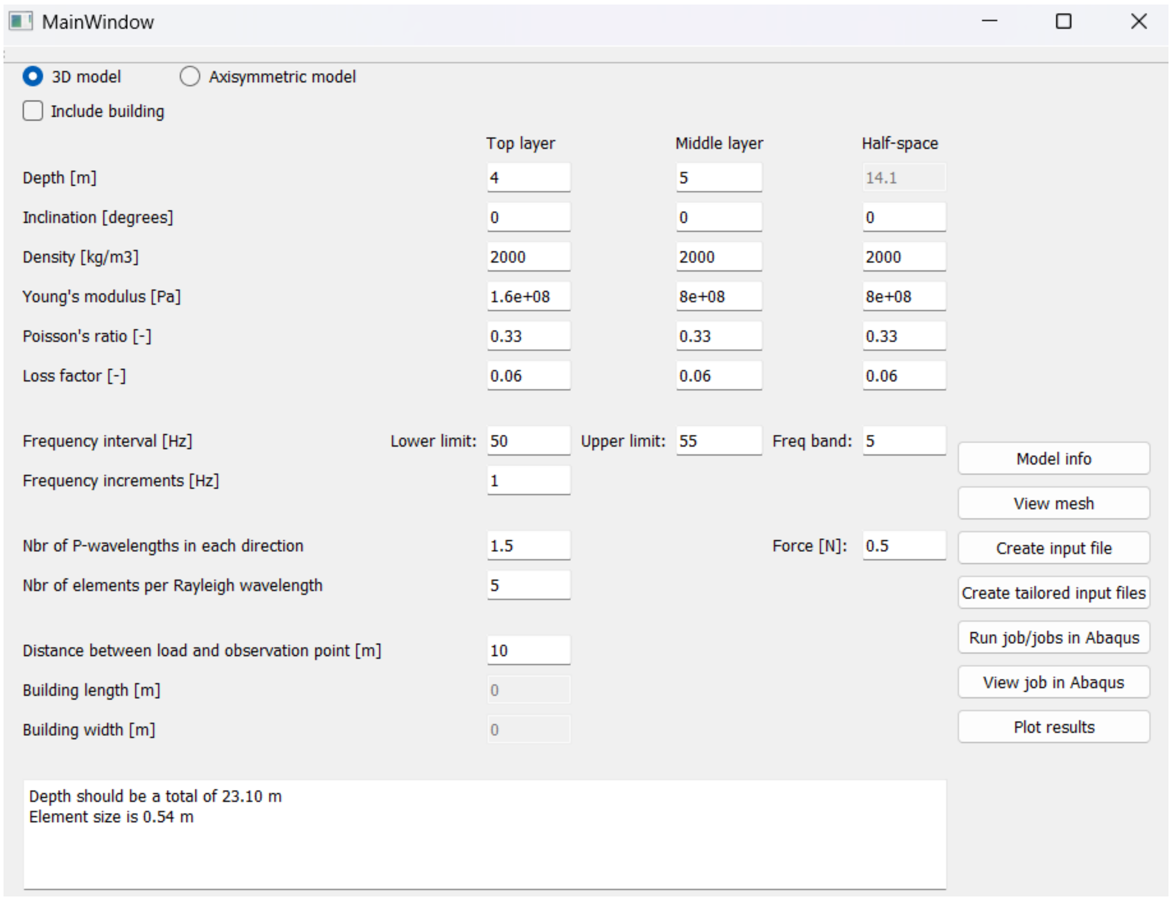
3.5 User Interface

A user interface for the program was created in a separate script, `mainprogram.py`, using the `QtPy` library and the design tool QtDesigner. This was done in order to improve user-friendliness and communication between the program and the user. A user manual for the interface can be found in Appendix B.

QtDesigner was used to design the complete layout of the interface, see Figure 3.7. The created ui-file containing the layout was imported to the script along with the script for the ground model. This allowed for the created user interface to be connected to the functionality of the ground model program.

The script contains two classes, `class MainWindow` and `class WorkerThread`. The functionality in the user interface is predominantly established in the `MainWindow` class, which manages all clicks and events from the user interface and sets up the majority of the connections between the QtDesigner file and the ground model program. The `WorkerThread` class is only used to execute time consuming operations using threads, i.e. parallel execution of a given code, in order to avoid the program being locked during these executions.

A lot of the choices that were made during the structuring of the program were done with the user interface in mind. For example, the depths of the layers were not managed thoroughly in the `BaseModel` class in favour of handling the total depth interactively in the user interface. This was achieved by having the third layer, i.e. the half-space, automatically update whenever a parameter affecting model dimensions is modified, as well as the interface generating a warning notification when the combined depth of the two overlaying layers exceeds the calculated optimal total depth.



[✓ 3D model] Axisymmetric model

☐ Include building

	Top layer	Middle layer	Half-space
Depth [m]	4	5	14.1
Inclination [degrees]	0	0	0
Density [kg/m3]	2000	2000	2000
Young's modulus [Pa]	1.6e+08	8e+08	8e+08
Poisson's ratio [-]	0.33	0.33	0.33
Loss factor [-]	0.06	0.06	0.06

Frequency interval [Hz]	Lower limit: 50	Upper limit: 55	Freq band: 5
Frequency increments [Hz]	1		
Nbr of P-wavelengths in each direction	1.5		Force [N]: 0.5
Nbr of elements per Rayleigh wavelength	5		
Distance between load and observation point [m]	10		
Building length [m]	0		
Building width [m]	0		

```
Depth should be a total of 23.10 m
Element size is 0.54 m
```

Figure 3.7: The user interface.

Additionally, the geometry of a model object was not created in the model's constructor directly by invoking its `geometry()` function, as this would cause the geometry of the model to be constructed directly upon starting the user interface and therefore causing the program to be slower. It would also cause problems with updating the model when a parameter was changed.

Furthermore, significant effort was put in to using the commands `gmsh.initialize()` and `gmsh.finalize()` appropriately, as using the latter inappropriately would terminate Gmsh and prevent further use of the program and using the former one inappropriately would cause an error if used before a previous Gmsh session has been finalized. The solution to this being that at the start of each new geometry creation, the program will finalize a current Gmsh session if there is one active, and then initialize a new session.

```python
def geometry(self):
    try:
        gmsh.finalize()
    except:
        pass
    gmsh.initialize()
```

However, this is a possibly flawed solution, as Gmsh will not be finalized for the last geometry created by the user and may lead to memory leaks.

There were also issues with using these commands in the `WorkerThread` class, as Gmsh is not designed for multi-threaded initialization. This resulted in the operations using Gmsh not being able to be executed from `WorkerThread`, most importantly the creation of input files, and the program therefore freezing when executing these operations.

As mentioned in Section 3.4.1, the use of the Abaqus Python library was made more difficult by the implementation of the user interface due to this library not supporting the QtPy library. This disrupted the initial plan of having the user being able to directly choose the odb-files to be plotted through the user interface. There are probably ways of solving this issue, but due to limited time these efforts were discarded.

Also due to limited time, the user interface lacks error handling leading the program to crash if invalid data types are entered as values, such as letters instead of numbers, or if unreasonable values are entered which produces either faulty or excessively large models. The user is therefore advised to enter only valid numerical values within reasonable ranges.

4 Results

Results are presented for both the axisymmetric model and the 3D model. All analyses were performed in the frequency domain. Although each analysis encompasses displacement, velocity, and acceleration, only acceleration data are presented. This choice is based on the main purpose of the study being parameter investigations and comparison with a semi-analytical model. Since displacement, velocity, and acceleration are mathematically related through time differentiation, presenting one variable was deemed sufficient for the comparative purposes of this study. The results for the axisymmetric model are presented for the horizontal and vertical directions, corresponding to the x-direction and the y-direction respectively, see Figure 4.1. The results for the 3D model are presented for horizontal and vertical directions also, corresponding to the x-direction and the z-direction respectively. The accelerations in the y-direction are not presented in the 3D model due to the observation point being located in the symmetry plane and the results in the y-direction therefore being zero, see Figure 4.2.

All analyses were conducted using the ground properties in Table 4.1. As the properties of the middle layer and the half-space were the same, they were essentially one and the same layer. Therefore the choice to resize the middle layer, if needed when model size decreases, was used. The size of the load was 1 N for the axisymmetric model and 0.5 N for the 3D model. The approximate distance between the load and the observation point was 10 m. For all analyses except for one including a building in Section 4.3, all layers had 0° inclination.

The results are compared to the semi-analytical Thomson-Haskell model to prove their validity.

Table 4.1: Properties of three ground layers.

Property	Top layer	Middle layer	Half-space
Depth (m)	4	-	-
ρ (kg/m^3)	2000	2000	2000
E (MPa)	160	800	800
ν	0.33	0.33	0.33
η	0.06	0.06	0.06

Figures 4.1 and 4.2 showcase examples of the two model types in Abaqus prior and after a steady state dynamics analysis. The most important take away from these being that the boundaries are largely non-reflective.

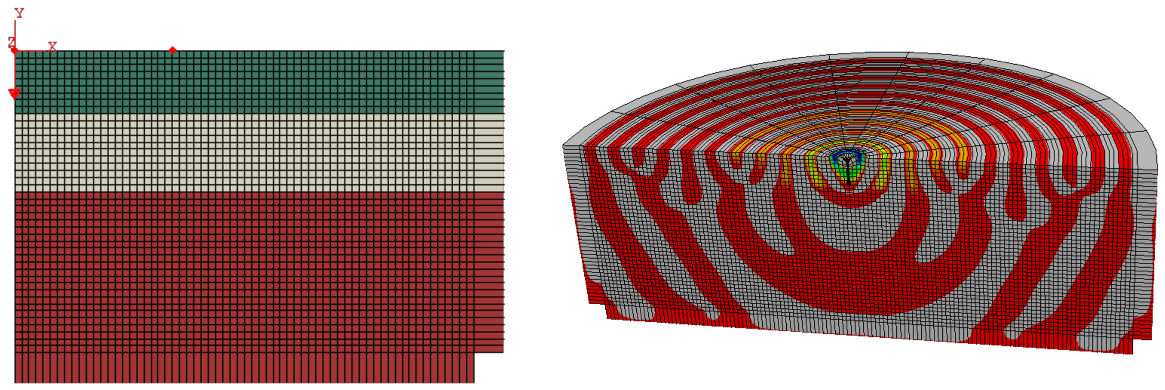


Figure 4.1: The axisymmetric model in Abaqus prior to and after analysis. The right model is swept 180° around the y-axis.

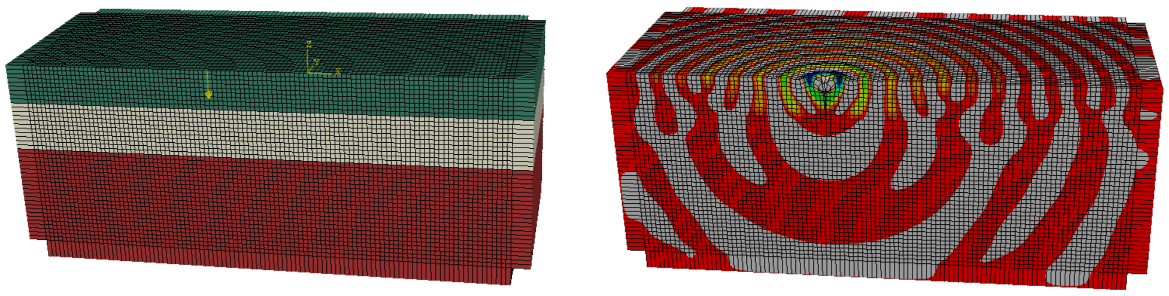


Figure 4.2: The 3D model in Abaqus prior to and after analysis.

There are significantly fewer nodes in the axisymmetric model compared to a 3D model with the same parameters. This resulted in the axisymmetric models being able to submit for analyses in Abaqus from a regular computer, while the 3D models required the use of a supercomputer. The computations of the 3D model were therefore enabled by resources provided by LUNARC, The Centre for Scientific and Technical Computing at Lund University.

The main issue when running the 3D model was its extensive use of RAM memory. Even on LUNARC, the 3D model with certain parameters was unable to run on regular nodes containing 256 GB RAM memory, as this resulted in out-of-memory errors. It was also difficult to gain more memory by running the model across multiple nodes, as the infinite elements prevented shared memory execution of the element calculations thus disabling parallel element operations. The solution to this was using nodes containing 512 GB RAM memory, these were however limited in number. The axisymmetric model was therefore studied first, often in the full 1-100 Hz interval, before determining a suitable smaller frequency range for the study of the 3D model.

4.1 Parameter studies

Parameter studies were conducted in order to investigate how different parameters affect the results and what values the parameters should have in order to produce sufficiently accurate results at a reasonable computational cost. The parameters that were studied included the number of P-wavelengths, the number of elements per Rayleigh wavelength, the size of the frequency increments and the size of the frequency bands.

4.1.1 Number of P-wavelengths

The number of P-wavelengths that is used affects the size of the model in accordance with Section 3.1. A study was done to determine the minimum number of P-wavelengths that provide sufficiently accurate results. The analysis was conducted using 5 elements per Rayleigh wavelength and 0.5 Hz frequency increments.

For the axisymmetric model, the 1-100 Hz interval was studied using 5 Hz sized frequency bands for 0.5, 1, 1.4, 1.5 and 2 P-wavelengths, see Figure 4.3.

The results from the study indicate that there is larger variations for the higher frequencies. The interval 60-75 Hz was therefore chosen to examine the parameter more closely, see Figure 4.4.

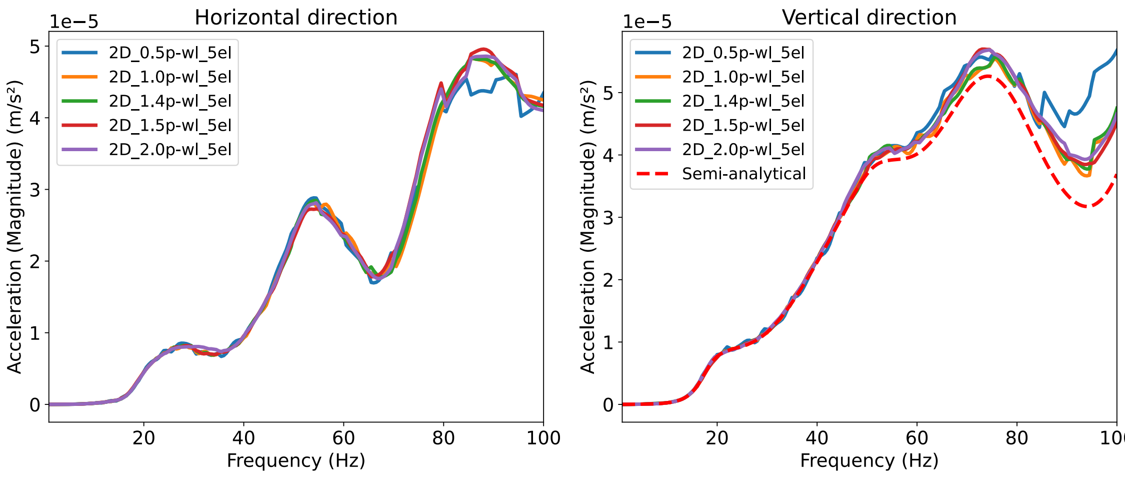


Figure 4.3: Accelerations in the axisymmetric model for different number of P-wavelengths in the 1-100 Hz interval.

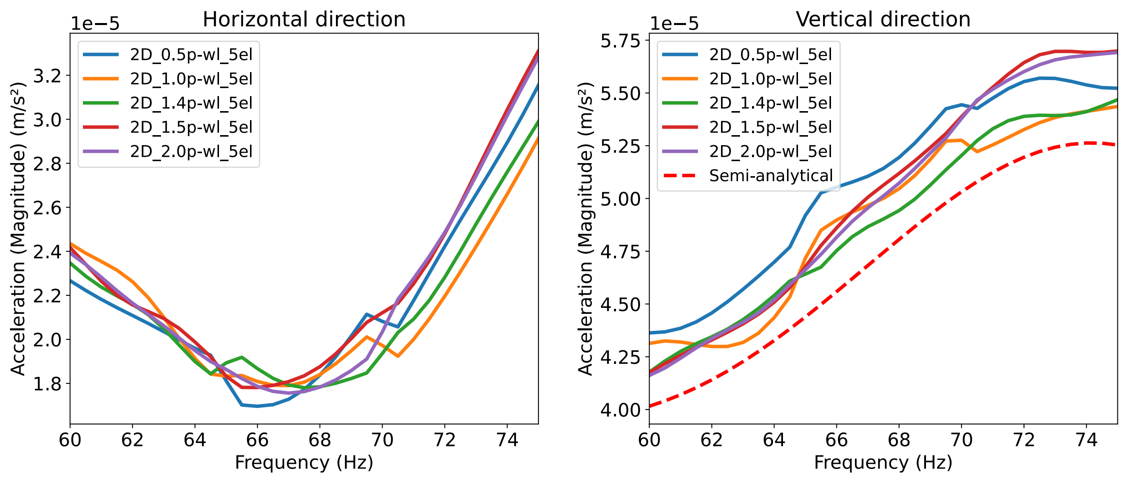


Figure 4.4: Accelerations in the axisymmetric model for different number of P-wavelengths in the 60-75 Hz interval.

The study was conducted for the 3D model using 1, 1.4, 1.5 and 2 number of P-wavelengths, see Figure 4.5. The models based on the three first number of P-wavelengths were able to run on the 256 GB nodes. However, the models using 2 number of P-wavelengths became very large. Even with the use of frequency bands the size of 2.5 Hz instead of 5 Hz, the models consisted of approximately 1.7 million nodes. Therefore, in addition to using 2.5 Hz frequency bands, the models had to be run on nodes containing 512 GB RAM memory, see Table 4.2.

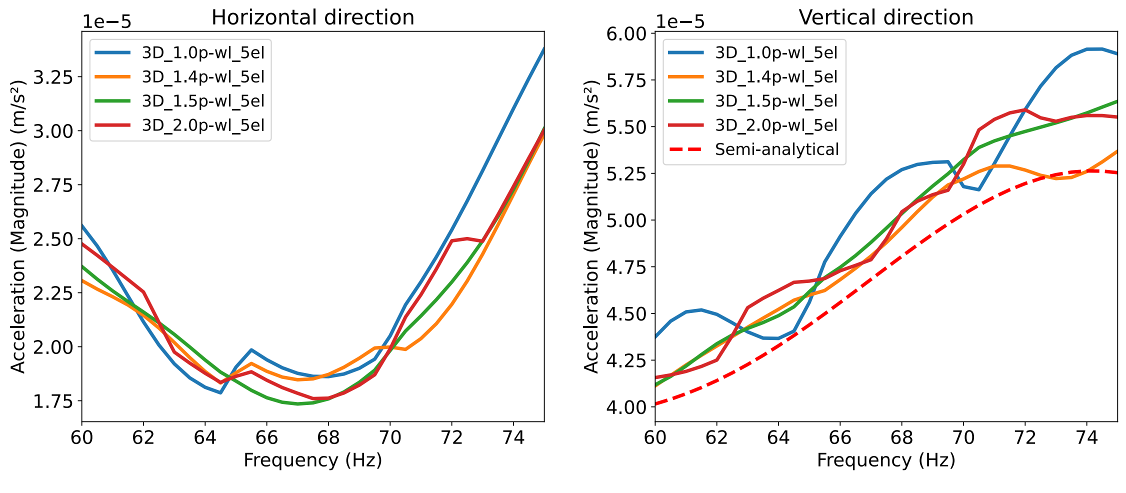


Figure 4.5: Accelerations in the 3D model for different number of P-wavelengths in the 60-75 Hz interval.

Table 4.2: Analyses times for models with different number of P-wavelengths in the 60-75 Hz interval.

	Axisymmetric model				
Nbr of P-wavelengths	0.5	1	1.4	1.5	2
Nbr of el per Rayl. wavelength	5	5	5	5	5
Frequency increments (Hz)	0.5	0.5	0.5	0.5	0.5
Frequency bands (Hz)	5	5	5	5	5
Nbr of inp files	3	3	3	3	3
Avg. nbr of nodes in inp files	2081	4958	8104	9019	14253
Time generating inp files (s)	0	0	1	1	1
Tot. sequential run time (min:s)	01:10	01:10	01:16	01:16	01:22
Parallel run time (s)	20	20	30	30	30
CPUs sequential/parallel	4/1	4/1	4/1	4/1	4/1

	3D model			
Nbr of P-wavelengths	1	1.4	1.5	2
Nbr of el per Rayl. wavelength	5	5	5	5
Frequency increments (Hz)	0.5	0.5	0.5	0.5
Frequency bands (Hz)	5	5	5	2.5
Nbr of inp files	3	3	3	6
Avg. nbr of nodes in inp files	302851	720933	858299	1867909
Time generating inp files (s)	11	23	28	59
Avg. run time per inp file (h:min:s)	00:52:22	05:16:27	09:16:54	13:00:43
Total run time (d-h:min:s)	02:37:06	15:49:20	1-03:50:41	3-06:04:16
CPUs	16	16	16	24
Utilized memory (GB)	75 of 249	238 of 249	238 of 249	497 of 497

The results in Figures 4.4 and 4.5 seem to indicate that both model types converge for 1.5 number of P-wavelengths. This value is therefore used for all following analyses. The results for the 3D model, Figure 4.5, are a bit curious however, as using 2 P-wavelengths seems to provide less accurate results compared to 1.5 P-wavelengths. The reason for this is not determined directly but may be due to smaller frequency bands being used, see Section 4.1.4.

4.1.2 Number of elements per Rayleigh wavelength

Another important factor for accuracy and computational efficiency is element size. In this section, different number of elements per Rayleigh wavelength will be investigated.

The frequency interval 60-75 Hz was used for the analyses of both models. The axisymmetric model was studied first, using 3, 4, 5, 6 and 10 number of elements per Rayleigh wavelength, see Figure 4.6. The 3D model was thereafter investigated with 3, 4, 5 and 6 elements per Rayleigh wavelength, see Figure 4.7. The 3D model was not analysed with 10 elements per Rayleigh wavelength as this produced models with approximately 6.400.000 number of nodes, which is too large to be analysed, even using the LUNARC nodes containing 512 GB RAM memory.

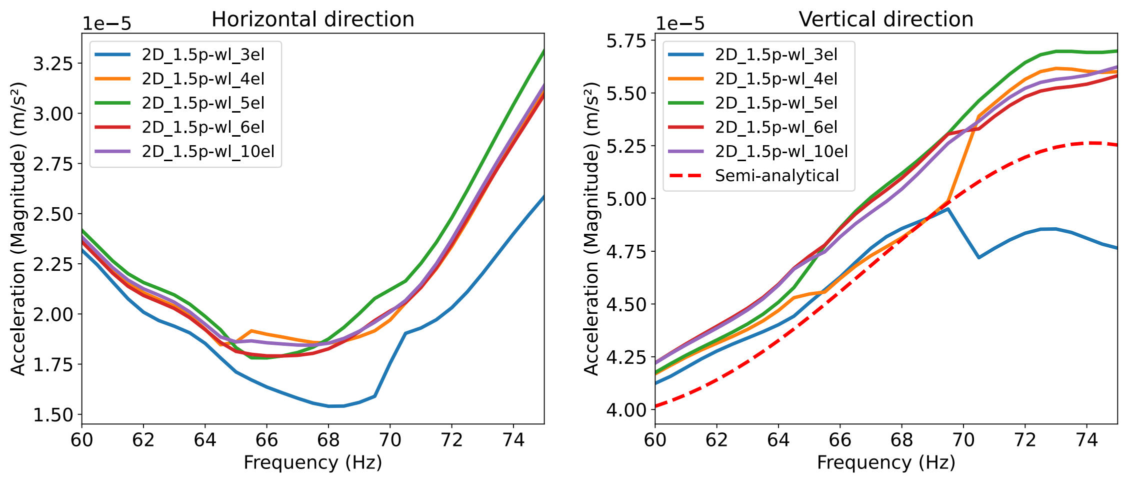


Figure 4.6: Accelerations in the axisymmetric model for different number of elements per Rayleigh wavelength in the 60-75 Hz interval.

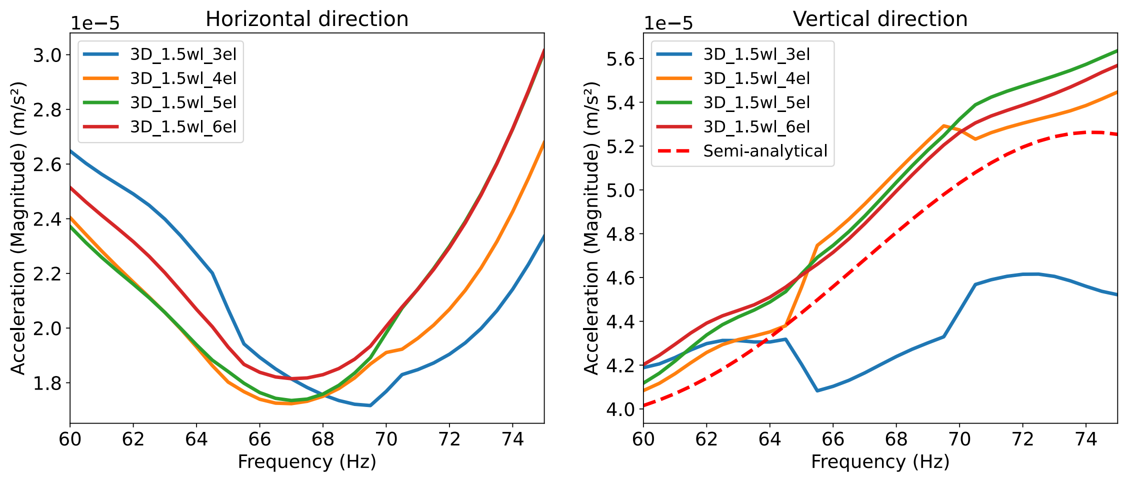


Figure 4.7: Accelerations in the 3D model for different number of elements per Rayleigh wavelength in the 60-75 Hz interval.

Figure 4.6 suggests that the use of 6 elements per Rayleigh wavelengths is sufficient for convergence. However, using 6 elements per Rayleigh wavelength in the 3D model generates models with approximately 1.460.000 nodes, while 5 elements per Rayleigh wavelength generates models with approximately 860.000 nodes, see Table 4.3. The decision was therefore made to use 5 elements per Rayleigh wavelength in the following analyses, as Figures 4.6 and 4.7 both show that this provides fairly accurate results while being significantly more computationally efficient, having a total run time of 28 hours using nodes with 256 GB RAM memory instead of 51 hours using nodes with 512 GB RAM memory, which are limited in number. Additionally, the 3D model with 6 elements was run using 24 CPUs instead of 16, requiring more Abaqus licenses and still taking almost twice as long compared to the 3D model with 5 elements.

Table 4.3: Analyses times for models with different number of elements per Rayleigh wavelength in the 60-75 Hz interval.

	Axisymmetric model				
Nbr of el per Rayl. wavelength	3	4	5	6	10
Nbr of P-wavelengths	1.5	1.5	1.5	1.5	1.5
Frequency increments (Hz)	0.5	0.5	0.5	0.5	0.5
Frequency bands (Hz)	5	5	5	5	5
Nbr of inp files	3	3	3	3	3
Avg. nbr of nodes in inp files	3521	6079	9102	12958	34042
Time generating inp files (s)	0	0	1	1	1
Tot. sequential run time (min:s)	01:10	01:10	01:16	01:22	01:46
Parallel run time (s)	25	30	30	30	50
CPUs sequential/parallel	4/1	4/1	4/1	4/1	4/1

	3D model			
Nbr of el per Rayl. wavelength	3	4	5	6
Nbr of P-wavelengths	1.5	1.5	1.5	1.5
Frequency increments (Hz)	0.5	0.5	0.5	0.5
Frequency bands (Hz)	5	5	5	5
Nbr of inp files	3	3	3	3
Avg. nbr of nodes in inp files	210567	465267	858299	1460021
Time generating inp files (s)	8	16	28	46
Avg. run time per inp file (h:min:s)	00:27:29	02:05:29	09:16:54	16:55:31
Total run time (d-h:min:s)	1:22:26	6:16:28	1-03:50:41	2-02:46:34
CPUs	16	16	16	24
Utilized memory (GB)	48 of 249	140 of 249	238 of 249	497 of 497

4.1.3 Frequency increments

The effects of different sized frequency increments was studied, again with the purpose of determining how computationally efficient the models can be without disrupting the results significantly. First, the axisymmetric model was studied for 0.25, 0.5 and 1 Hz increments in the full 1-100 Hz range, see Figure 4.8. Then the axisymmetric and 3D model were analysed respectively in the 60-75 Hz range, see Figures 4.9 and 4.10.

All analyses were performed using 1.5 P-wavelengths determining model size, 5 elements per Rayleigh wavelength determining element size and frequency bands the size of 5 Hz. As the model- and element sizes are independent of the size of the frequency increments, the number of nodes and the time it took to generate the input files was not examined, see Table 4.4.

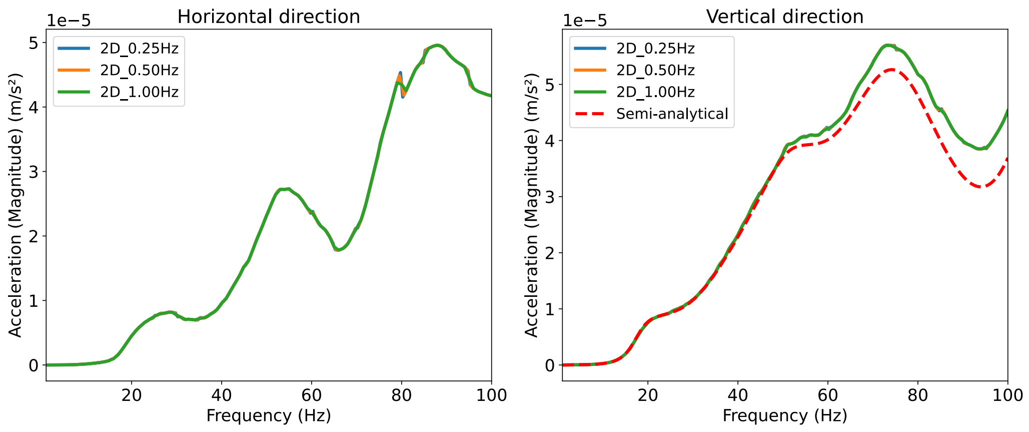


Figure 4.8: Accelerations in the axisymmetric model for different sized frequency increments in the 1-100 Hz interval.

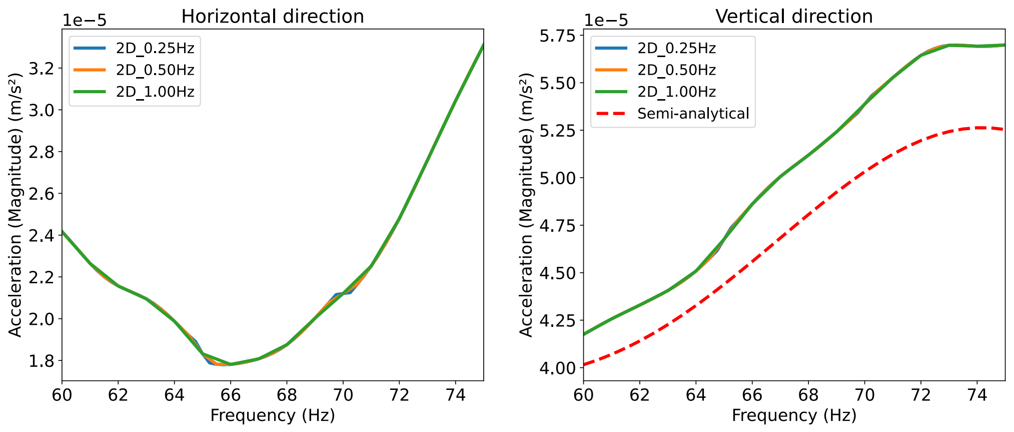


Figure 4.9: Accelerations in the axisymmetric model for different sized frequency increments in the 60-75 Hz interval.

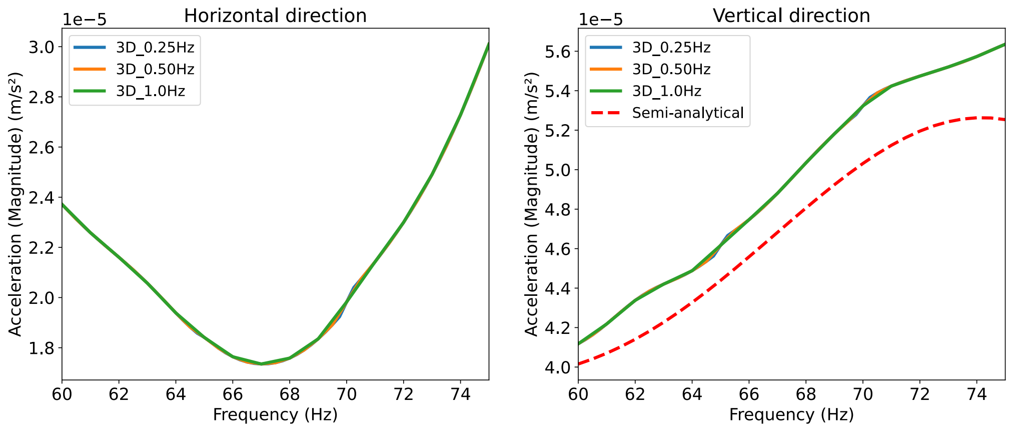


Figure 4.10: Accelerations in the 3D model for different sized frequency increments in the 60-75 Hz interval.

Table 4.4: Analyses times for models with different sized frequency increments.

	Axisymmetric model		
Frequency increment (Hz)	0.25	0.50	1.00
Nbr of inp files	3	3	3
Tot. sequential run time (min:s)	01:22	01:16	01:10
Parallel run time (s)	35	30	25
CPUs sequential/parallel	4/1	4/1	4/1

	3D model		
Frequency increment (Hz)	0.25	0.50	1.00
Nbr of inp files	3	3	3
Avg. run time per inp file (h:min:s)	17:26:40	09:06:20	04:19:06
Total run time (d-h:min:s)	2-04:19:59	1-03:19:00	12:57:18
CPUs	16	16	16
Memory utilized (GB)	239 of 249	239 of 249	239 of 249

The results from both model types show that the size of the frequency increments ranging between 0.25-1 Hz only affects the results minimally, and for the cases that it does affect the results it seems to be where there is an overlap in input files, most apparent in the left plot in Figure 4.9 at 65 and 70 Hz. Even for the frequencies in the lower ranges, the results do not seem to differ much, as indicated by Figure 4.8.

For the 3D model, the Abaqus analysis time is halved when 1 Hz increments are used in favour of 0.5 increments, leading to the decision to use 1 Hz increments in the following analyses. However, it should be noted that a finer frequency incrementation can be needed if evaluating the response in a building positioned on top of the ground.

4.1.4 Frequency bands

As discussed in Section 2.1.1, the size of the frequency band as well as where in the frequency range the band is located will affect the relationship between model- and element size. A smaller frequency band will generate models with a coarser mesh, while a larger band will generate a finer mesh. The same is true for the location of the band, where bands located in the lower frequencies will yield a very fine mesh while the same sized bands in the upper frequencies will yield a coarser mesh.

The relationship between model- and element size will determine the number of nodes in a model, see Figure 4.11. This in turn affects the results, providing more accurate results when using a finer mesh while also being more computationally expensive.

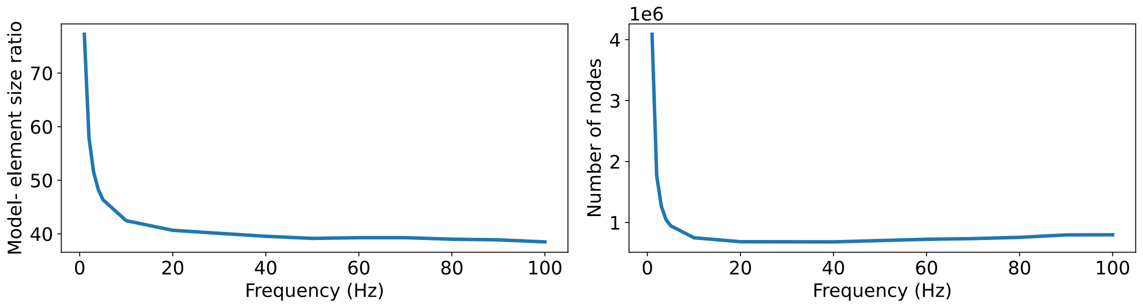


Figure 4.11: Ratio between model- and element size and number of generated nodes in the 3D model for a 1 Hz frequency band in the 1-100 Hz interval.

The size of the frequency bands will also affect the number of interpolations that are done with the data, as explained in Section 3.4.2. Some of the previous results seem to indicate that this has the effect of making the diagram jerk where an interpolation was made. This was briefly mentioned in the previous section and when discussing Figure 4.5. In this section, the effect of different sized frequency bands will be studied further.

The idea was at first to use a logarithmic scale when dividing the interval 1-100 Hz into frequency bands, starting with the smallest band at 1 Hz and increasing the band size logarithmically. Using 15 logarithmically placed points with an approximate growth factor of 1.39 gives frequency bands with node counts as shown in Table 4.5. This gives a good division of nodes among the frequency bands. However, it is not very computationally efficient. All the frequency bands require LUNARC nodes with 512 GB RAM memory to analyse, and as there is not a large number of these the queuing time for each job to start is extensive.

Furthermore, as frequency increments of 1 Hz were used in all frequency bands except for the bands smaller than 1 Hz, in which case 2 steps were used instead, the large frequency bands had a significantly longer running time due to there being more steps in the analysis. Logarithmic scaled frequency increments were not implemented due to Section 4.1.3 indicating that 1 Hz increments were sufficient, even for the smaller frequencies. The band 73-100 Hz would therefore take approximately 4 days to analyse while the 1-1.39 band would take approximately 5 hours using 16 and 20 CPUs respectively, see Table 4.8.

Table 4.5: Logarithmic sized frequency bands and the number of nodes generated for the 3D model.

Frequency band	Number of nodes	Frequency band	Number of nodes
1-1.39	1424875	10.03-13.94	1463441
1.39-1.93	1424875	13.94-19.37	1476092
1.93-2.68	1412837	19.37-26.93	1514045
2.69-3.73	1412837	26.93-37.43	1579363
3.73-5.19	1425488	37.43-52.02	1615253
5.19-7.21	1425488	52.02-72.31	1703810
7.21-10.03	1438139	72.31-100	1817669

Instead of using a logarithmic scale, the division of frequency bands was done manually by balancing the node count with an estimated guess at analysis time based on the number of steps in the analysis, attempting to produce approximately the same analyses times for all frequency bands. Another aspect that was taken into consideration was having the majority of the bands being able to analyse on the 256 GB nodes. This produced more input files that had to be submitted for analyses but on the other hand allowed for the possibility of parallel running while also providing significantly faster analyses.

A study was conducted to determine what sized frequency bands would be appropriate for both accuracy and efficiency. For this purpose, the axisymmetric model with differently sized frequency bands was studied in the full 1-100 Hz interval, see Figure 4.12, and both model types were then analysed in the 60-100 Hz range, see Figures 4.13 and 4.14 as well as Table 4.6.

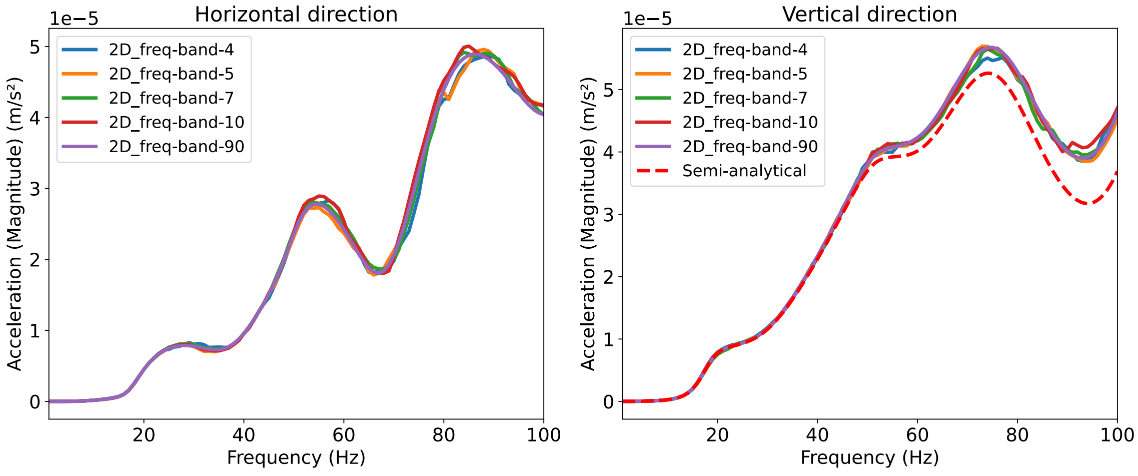


Figure 4.12: Accelerations in the axisymmetric model for different sized frequency bands in the 1-100 Hz interval.

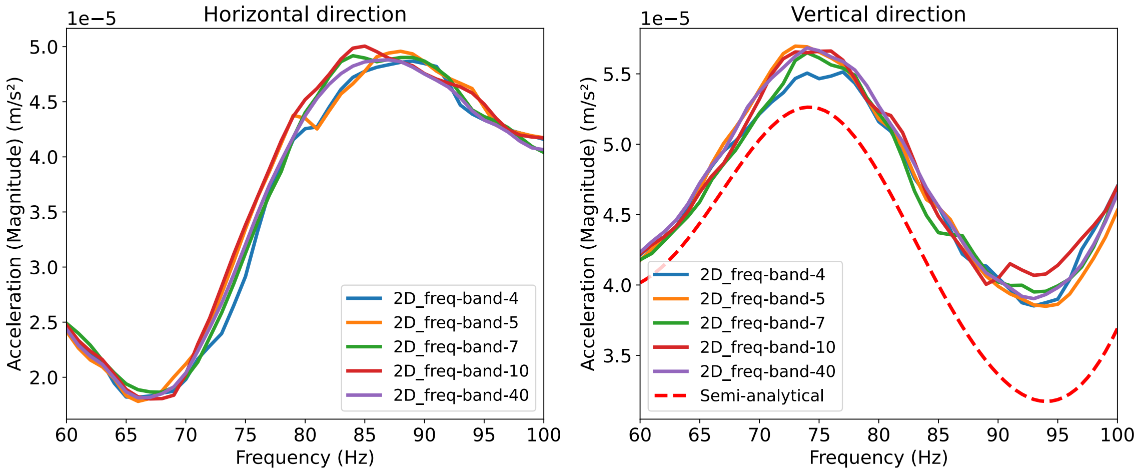


Figure 4.13: Accelerations in the axisymmetric model for different sized frequency bands in the 60-100 Hz interval.

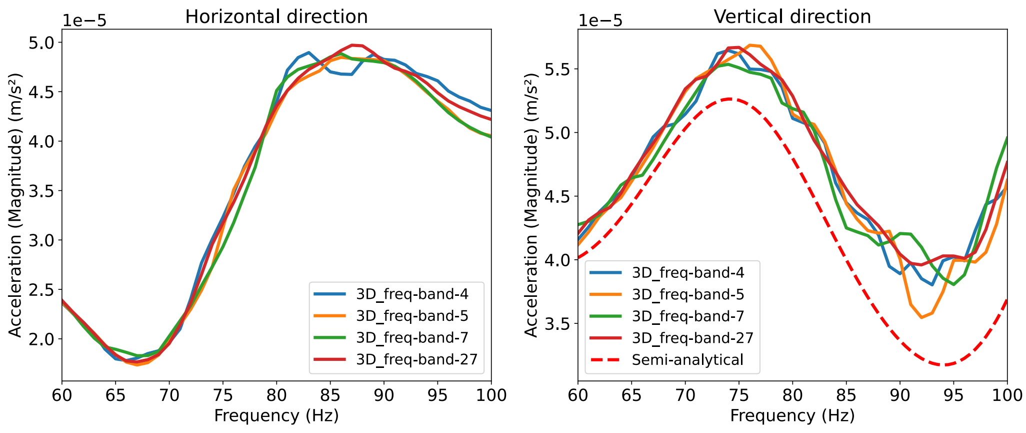


Figure 4.14: Accelerations in the 3D model for different sized frequency bands in the 60-100 Hz interval.

Table 4.6: Analyses times for different sized frequency bands in the 60-100 Hz interval.

	Axisymmetric model				
Frequency band size (Hz)	4	5	7	10	40
Nbr of inp files	10	8	6	4	1
Avg. nbr of nodes in inp files	9228	9443	9797	10520	20229
Time generating inp files (s)	2	2	1	1	0
Avg. run time per inp file (min:s)	00:23	00:23	00:26	00:25	00:50
Tot. sequential run time (min:s)	03:53	03:06	02:33	01:42	00:50
Parallel run time (min:s)	01:20	00:55	01:00	00:30	-
CPUs sequential/parallel	4/1	4/1	4/1	4/1	4/-

	3D model			
Frequency band size (Hz)	4	5	7	27
Nbr of inp files	10	8	6	1
Avg. nbr of nodes in inp files	843083	871893	936559	1741368
Time generating inp files (min:s)	01:32	01:14	00:59	00:19
Avg. run time per inp file (d-h:min:s)	03:28:07	04:21:45	06:56:49	3-12:48:36
Tot. sequential run time (d-h:min:s)	15:23:03	1-10:53:59	1-23:05:34	3-12:48:36
CPUs	16	16	16	16
Memory utilized (GB)	239 of 249	239 of 249	239 of 249 (OoM)	497 of 497

It is difficult to draw any definite conclusions from the results. For example, the 5 Hz band seems to converge for the axisymmetric model in the vertical direction, Figure 4.13, while the 10 Hz band deviates in the 90-95 Hz interval. However, in the horizontal direction, the 5 Hz band deviates at 80 Hz while the 7 Hz band converges. In the 90-95 Hz interval, the 7 Hz band converges towards the 40 Hz band while the 10 Hz band does not.

The same discrepancies can be observed for the 3D model, Figure 4.14. In the horizontal direction, the 5 Hz band seems to be converging while in the vertical direction it seems to be deviating even more than the 4 Hz band in the 85-95 Hz interval. The only feasible conclusion being that for higher frequencies, from 80 Hz and upwards, the results become more uncertain, for both model types.

For one of the 3D models in the 7 Hz band, specifically the 65-72 Hz interval, an out-of-memory error was received from Abaqus at the 70 Hz step after running for 5.5 hours, see Table 4.6. This makes the 7 Hz band unreliable to analyse on the 256 GB nodes. This, together with the fact that the results in Figures 4.13 and 4.14 were a bit ambiguous for all bands but seem to generate sufficiently accurate results for the 5 Hz bands, at least up to 80 Hz, lead to the conclusion that the 5 Hz band was the most practical choice and is therefore used in the following analyses.

4.2 Model validation and efficiency

Based on the results from the parameter studies, the values 1.5 P-wavelength, 5 elements per Rayleigh wavelength, 1 Hz increments and 5 Hz bands were used to analyse the 1-100 Hz interval for both the axisymmetric model and the 3D model. These were then compared to the semi-analytical model with the purpose of validating the accuracy of the created models in the 1-100 Hz interval, Figures 4.15 and 4.16.

The computational time required to analyse the 1-100 Hz range is documented in Table 4.7 for the axisymmetric model and Table 4.8 for the 3D model. The full 1-100 Hz range produces 45 million nodes for the axisymmetric model, while the number of nodes for the 3D model in the full 1-100 Hz range is unknown, as the program crashes when trying to produce the mesh. Suffice to say the node count is significantly higher than 45 million for the 3D model in the full 1-100 Hz range. The conclusion was therefore drawn that without some division of the models, the full 1-100 Hz range cannot be analysed using the instruments available for this project.

Due to this fact, the axisymmetric model was divided into 1-10 Hz and 10-100 Hz bands, which were used to compare computational time with the axisymmetric model divided into 5 Hz bands. The 3D model had to be divided further, and was done so using the bands 36-52 Hz, 52-73 Hz and 73-100 Hz. These were then compared to the 3D model using 5 Hz bands. Beneath 40 Hz, smaller frequency bands were used to avoid out-of-memory Abaqus errors as well as the results from Section 4.1.4 indicating that it is mainly for the higher frequencies that band size affects the results.

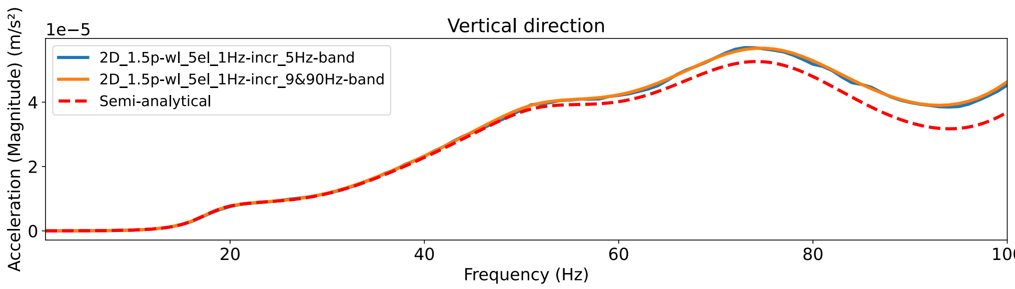


Figure 4.15: Comparison of accelerations in tailored axisymmetric models using 5 Hz bands and less tailored models.

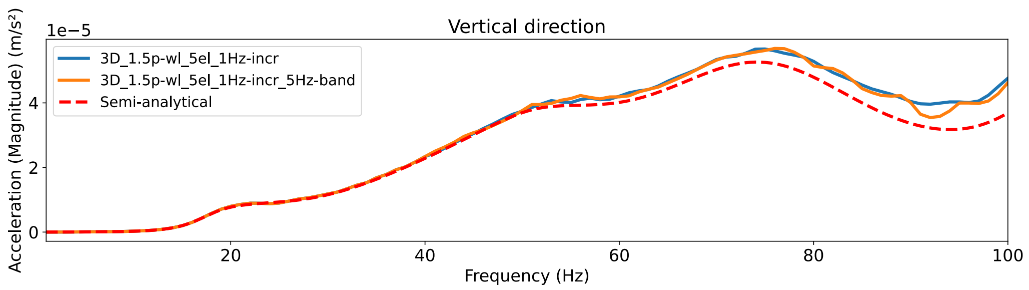


Figure 4.16: Comparison of accelerations in tailored 3D models using 5 Hz bands above frequencies of 40 Hz and less tailored models.

Table 4.7: Analyses times for different sized axisymmetric models.

	5 Hz band		
Freq. band (Hz)	Nbr of nodes	Run time (min:s)	CPUs
1.0-5	115440	00:25	4
5.0-10	20002	00:27	4
10.0-15	11960	00:23	4
15-20	9946	00:23	4
20-25	9090	00:24	4
25-30	8914	00:23	4
30-35	8386	00:24	4
35-40	8529	00:23	4
40-45	8490	00:23	4
45-50	8445	00:24	4
50-55	8582	00:24	4
55-60	9050	00:23	4
60-65	8796	00:23	4
65-70	9064	00:23	4
70-75	9198	00:23	4
75-80	9332	00:23	4
80-85	9600	00:23	4
85-90	9734	00:23	4
90-95	9778	00:23	4
95-100	10040	00:23	4
Tot. sequential run time:		**08:16**	
Parallel run time:		02:31	1

	Larger sized bands		
Freq. band (Hz)	Nbr of nodes	Run time (min:s)	CPUs
1-10	457855	05:44	4
10-100	492925	30:40	4
Tot. sequential run time:		**36:24**	
Parallel run time:		30:40	4

Computational time difference for sequential run (min:s):	**28:08**
Computational time difference for parallel run (min:s):	**28:09**

Table 4.8: Analyses times for different sized 3D models.

	5 Hz band			
Freq. band (Hz)	Nbr of nodes	Run time (h:min:s)	CPUs	Memory (GB)
1-1.5	1765805	05:27:34	20	512
1.5-2	1255248	02:34:30	20	512
2.0-3.0	1765805	05:44:50	20	512
3.0-4.0	1267015	02:37:49	20	512
4.0-6.0	1780559	08:56:20	20	512
6.0-8.0	1302155	04:11:41	20	512
8.0-11	1400353	06:42:23	20	512
11.0-14	1131533	04:03:17	20	512
14.0-16	862730	02:23:07	16	256
16.0-18	835525	02:15:12	16	256
18.0-20	782983	01:51:23	16	256
20-23	889505	03:34:00	16	256
23-26	852215	03:03:53	16	256
26-29	825601	02:56:53	16	256
29-32	807538	02:57:20	16	256
32-36	869317	04:00:00	16	256
36-40	840278	**03:48:11**	16	256
40-45	894970	**05:05:54**	16	256
45-50	864833	**04:23:43**	16	256
50-55	873018	**04:31:44**	16	256
55-60	900482	**05:02:55**	16	256
60-65	850472	**04:17:37**	16	256
65-70	858299	**04:18:32**	16	256
70-75	866126	**04:17:31**	16	256
75-80	881780	**04:28:35**	16	256
80-85	889607	**04:27:11**	16	256
85-90	876902	**04:19:17**	16	256
90-95	872242	**04:13:51**	16	256
95-100	879719	**04:31:25**	16	256
36-100 Hz total run time:		**2-9:46:26**		

	Larger sized bands			
Freq. band (Hz)	Nbr of nodes	Run time (d-h:min:s)	CPUs	Memory (GB)
36-52	1786507	2-08:25:56	16	512
52-73	1778075	3-02:07:59	16	512
73-100	1741368	3-23:29:12	16	512
36-100 Hz total run time:		**9-3:42:19**		

Computational time difference for sequential run (d-h:min:s):	**6-18:5:53**

The results in Tables 4.7 and 4.8 show that the computational time difference is significant. The axisymmetric model is analysed 4.4 times faster, taking approximately

8.5 minutes instead of 36.5 minutes, when using sequential analyses and 12.2 times faster, taking approximately 2.5 minutes instead of 30.5 minutes, when implementing parallel analyses. The 3D model is analysed 3.8 times faster in the 36-100 Hz interval when comparing sequential analyses times, taking approximately 2 days and 10 hours instead of 9 days and 4 hours. When applying parallel analyses, the 75-100 Hz range could be analysed in approximately 4.5 hours instead of almost 4 days.

4.3 Added building and inclined ground layer

To showcase an application of the program, a building was included to the 3D ground model, Figure 4.17. The model was analysed with no inclination in either layer, and then with an inclination of 2 degrees in the middle layer, see Figure 4.18.

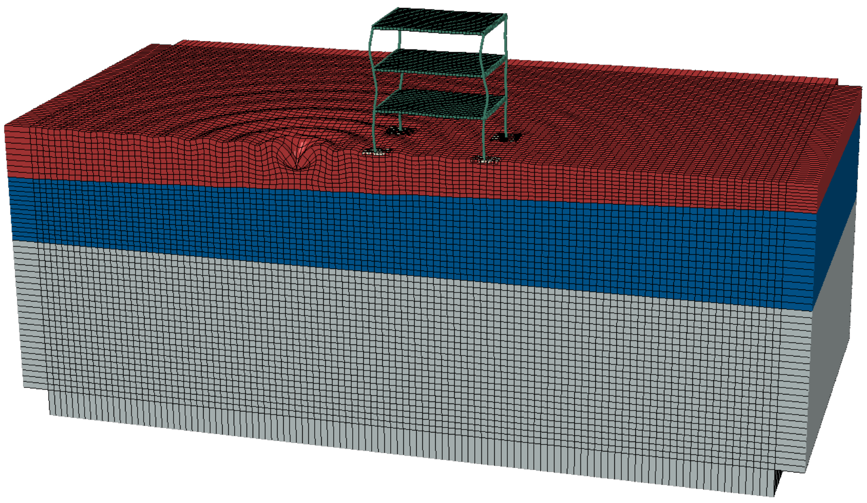


Figure 4.17: The 3D ground model with a building included and an inclined middle layer.

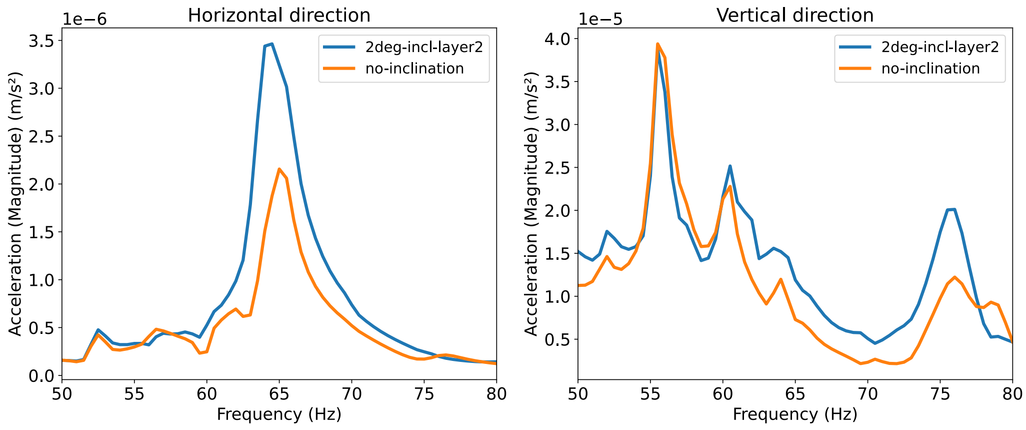


Figure 4.18: Accelerations in the middle of the building's first floor slab.

Figure 4.18 indicates that the inclined middle layer causes there to be higher reflections in the observation point due to the boundary to the second layer being closer.

5 Discussion

The aim of the dissertation was to develop a Python program that generates efficient ground models for Abaqus analysis of ground vibrations. To achieve this goal, three main aspects were addressed.

First, the models were created within the framework established by Abaqus, ensuring compliance with requirements for node ordering, sweep direction, and input file formatting. Second, appropriate model- and element dimensions that would achieve both sufficient accuracy and computational efficiency were established through systematic parameter studies, where parameters affecting model- and element size were analysed and optimal values were decided. Finally, model validation and efficiency assessment were conducted. The models were validated by comparing results against an established semi-analytical model for traffic-induced vibrations. Computational efficiency was evaluated by comparing analysis times between the tailored models and partially tailored reference models, as fully untailored models would have been too computationally demanding to analyse.

The following sections discuss each of these aspects in detail, analysing the findings and their implications for ground modelling in traffic-induced vibration analyses.

5.1 Program development

While the main functionality of the program was the creation of the models and making them compatible with Abaqus, additional features were developed to streamline the analysis process. The program includes functionality for submitting jobs directly to Abaqus, with the possibility to submit multiple analyses simultaneously if desired. Post-processing was also integrated into the program to enable faster and easier plotting of results. To make the program more user-friendly, a graphical user interface was created to manage all operations.

The biggest challenges were faced during different stages of development, but particularly when achieving Abaqus integration. The input file structure was especially challenging, as one keyword placed in the wrong line or data not being structured correctly would cause the analysis to fail in Abaqus. Further complexity was added when the option to include a building in the 3D model was implemented, which required the use of instances to avoid conflicts in node numbering. This approach demanded a different structure for keyword placements, with some element sets needing to be defined outside of the instance definitions and some inside, depending on their usage.

Getting the mesh right for structured quadrilateral elements was also challenging until the right tools in Gmsh were found, specifically the Transfinite functions. For a line, this function takes the number of nodes to be placed on the line as an argument, see

Section 2.3.2. Therefore, the element size cannot be determined exactly by the number of elements per Rayleigh wavelength. If the line divided by the calculated element size is not an integer, which it most probably will not be, the number of nodes is rounded upwards, making the mesh slightly finer than calculated. This is not a problem per se as a finer mesh is generally better for accuracy, but it is worth mentioning that the number of elements per Rayleigh wavelength determined in the parameter study should be considered more as a guideline than an absolute specification.

Furthermore, loading and observation points were approximated by finding the node closest to the wanted x-coordinate instead of creating exact geometric points. This was done as it was deemed unnecessarily complicated to create precise points during the geometry stage, both regarding the manual geometry operations, see Section 2.3.2, and because it would complicate the extrusion process for infinite domains. The approximation method was considered acceptable since the deviation is usually only a few centimetres and was not expected to affect the results significantly.

Additional challenges were mainly related to the user interface implementation, as discussed in Section 3.5. These issues primarily involved restructuring function calls within the program to ensure proper operation through the interface. For instance, the program had to be modified to avoid creating models automatically when starting, as well as to recalculate model and element dimensions when parameters are changed. A particularly frustrating limitation was the inability to implement direct selection of odb-files for result plotting through the user interface, which was caused by conflicts with the Abaqus Python library.

5.2 Model tailoring through parameter studies

The parameters that were studied included the number of P-wavelengths used to define model size, the number of elements per Rayleigh wavelength used to determine element size, the size of the frequency increments which affects analysis time but not model dimensions, and finally the size of the frequency bands.

The axisymmetric model proved useful as a guide for expected results while providing fast analyses, which helped identify areas to focus on for the 3D analyses that were significantly more computationally expensive. It is worth noting that the axisymmetric model represents a half-sphere when swept around the y-axis, see Figure 4.1, while the 3D model contain edges where there are no infinite elements, see Figure 4.2, which may lead to some reflections from these areas.

The number of P-wavelengths used for defining model dimensions that provided accurate results while also being a feasible option computationally wise was found to be 1.5, as this value converged when compared to 2 P-wavelengths in the axisymmetric model, Figure 4.4, and also followed the semi-analytical data well for the 3D model, Figure 4.5. Using a larger value proved difficult for the 3D model as it created significantly larger models that required a decrease in frequency band size to 2.5 Hz, which may have caused the curve to be less smooth.

For the number of elements per Rayleigh wavelength, 5 was considered to provide acceptable results, even though proper convergence was not achieved. 6 elements appeared to converge better when compared to 10 elements in the axisymmetric model, Figure 4.6, but the computational cost of using 6 elements in the 3D model was too high, taking more than twice the analysis time and requiring 512 GB of memory to run. The 3D results, Figure 4.7, indicate that using 5 elements is not too significant a reduction, with the curve deviating only slightly from the 6-element case.

The frequency increment study yielded quite straightforward results, demonstrating that 1 Hz increments produced nearly identical outcomes to 0.25 Hz or 0.5 Hz increments for both models, Figures 4.9 and 4.10. This consistency was observed across the full 1-100 Hz frequency range for the axisymmetric model. Due to computational constraints, the complete frequency range was not examined for the 3D model. However, given the nearly identical behaviour observed between both models within the 60-75 Hz range, it was assumed that the 3D model response would remain consistent with the axisymmetric model across the remaining frequency interval. While this assumption appears reasonable, it should be acknowledged as a limitation of the study. The implementation of 1 Hz increments resulted in a computational time reduction of over 50% for the 3D model compared to 0.5 Hz increments, making this the preferred choice from an efficiency perspective.

The study of frequency band size was more extensive, as it affected results in two ways: through interpolation of overlapping data when combining multiple input files, and due to changing relationships between model- and element dimensions. Another key consideration was dividing the frequency bands into sizes that could be analysed using 256 GB of RAM while balancing node count with the number of analysis steps, attempting to achieve approximately equal computational times across the 1-100 Hz interval.

It was quite difficult to draw definitive conclusions from the results, as convergence for a certain frequency band size appeared to be achieved in one direction while deviating in another. For example, the 5 Hz band in the 3D results, Figure 4.14, or cases where larger bands provided less accurate results than smaller ones, such as the 10 Hz band compared to the 7 Hz band in the axisymmetric results, Figure 4.13. The only clear conclusion was that frequency band size affects results most significantly at higher frequencies, indicating that larger model dimensions are needed for these frequencies. The most optimal convergence would naturally be achieved using the 27 Hz band provided in the 3D model, but this would defeat the purpose of tailored models since this represents the largest model that can be analysed using 512 GB of memory. The 7 Hz band would be the next logical choice, but since this caused memory errors when running on 256 GB, it was rejected in favour of the 5 Hz band. This provided accurate results up to approximately 80 Hz, after which the results deviate somewhat.

5.3 Validation and efficiency assessment

After the parameter study, the models were validated across the 1-100 Hz range using 1.5 P-wavelengths, 5 elements per Rayleigh wavelength, 1 Hz increments, and 5 Hz frequency bands by comparing them to the semi-analytical data.

Both model types seem to be deviating somewhat from the semi-analytical data at 50 Hz and upwards. The reason for this was not able to be established, but may be due to the FE models solving the system of equations in a different manner than the semi-analytical model. Further investigations would have to be conducted to establish the exact reason.

However, the results from the axisymmetric model, Figure 4.15, show that the tailored models follow the shape of the curve for the semi-analytical data accurately even when exhibiting higher accelerations for frequencies above 50 Hz. This may be due to the axisymmetric model acting as a perfect half-sphere, as discussed in Section 4, leading the vibrations to hit the boundary of the infinite elements in an exact perpendicular angle and therefore causing minor boundary reflections.

A comparison between the tailored models and the partially tailored ones, which consisted of two models in the bands 1-10 Hz and 10-100 Hz respectively, show that the tailored models are significantly faster, taking 2.5 minutes when run in parallel and approximately 8 minutes when run sequentially, compared to 30.5 min for parallel analyses and 36.5 minutes for sequential analyses of the two less tailored models.

The 3D results, Figure 4.16, provided a slightly less accurate shape of the curve when compared to the semi-analytical model for higher frequencies. This was expected for several reasons, one being that the number of elements per Rayleigh wavelength and the size of the frequency bands had to be chosen in a manner that would allow for faster analyses, and the other reason being that the 3D model had edges were vibrations would not hit the boundaries to the infinite elements in a perpendicular angle, causing there to be more reflections at these boundaries. Nevertheless, the results are considered reasonably accurate when compared to the semi-analytical data.

The computational time difference is significant for the 3D model, with the ability to run the 36-100 Hz interval in 2 days and 10 hours when run sequentially, compared to 9 days and 4 hours for the less tailored models. When models are run in parallel, the computational time difference becomes even more significant, analysing the 75-100 Hz range in 4.5 hours instead of 4 days.

6 Conclusions

Based on the aims and discussion of the dissertation, a number of conclusions can be drawn. These are listed in bullet points below.

Accurate ground modelling using Python while maintaining Abaqus compatibility:

- A comprehensive Python program was successfully developed that generates Abaqus-compatible ground models for traffic-induced vibration analysis
- Key compatibility challenges were overcome, including proper node ordering, sweep direction requirements, and input file formatting according to Abaqus framework
- Implementation of instances for building inclusion required restructuring of keyword placements but was successfully achieved
- Approximation methods for loading- and observation points by finding nearest nodes proved acceptable with negligible deviations

Parameters relevant for accurate and computationally efficient ground modelling:

- 1.5 P-wavelengths was identified as the optimal model size, providing sufficient convergence while maintaining computational feasibility
- 5 elements per Rayleigh wavelength was determined to be acceptable, balancing accuracy with computational cost as 6 elements per Rayleigh wavelength required 512 GB memory and doubled analysis time
- 1 Hz frequency increments produced nearly identical results to smaller increments while reducing computational time by over 50%
- 5 Hz frequency bands provided the best compromise between accuracy and memory constraints, with adequate results up to approximately 80 Hz

Time savings through tailored models:

- Tailored axisymmetric models achieved significant time reductions: 8 minutes vs 36.5 minutes for sequential analyses and 2.5 minutes vs 30.5 minutes for parallel analyses in the 1-100 Hz interval
- 3D model efficiency gains were substantial: 2 days 10 hours vs 9 days 4 hours for sequential analysis in the 36-100 Hz interval and 4.5 hours vs 4 days for analyses in the 75-100 Hz interval

7 Further studies

This project could be further improved in several aspects. These include aspects related to the user interface, further implementations of the ground models as well as further development of the program's functionality. Examples of possible further studies are listed in bullet points below.

- Error handling in the user interface as well as improving layout and functionality, for example by including the option of uploading or downloading parameters from/to file
- Extracting data from odb-files directly from the user interface
- Providing the option of adding a tunnel or an underground basement in the ground model
- Making the program compatible with LUNARC by adding functionality such as writing Bash scripts with user-specified memory and CPU usage
- Submitting jobs to the SLURM queue directly through the user interface, either sequentially or in parallel, with jobs being submitted automatically when previous ones are finished after a control has been made of available Abaqus licenses
- Automatizing the size of the frequency bands based on the frequency interval studied
- Investigating the cause for the deviation between the FE models and the semi-analytical model for frequencies above 50 Hz
- Examining the impact of the depth of the infinite elements
- Including compatibility with other FE softwares, for example Ansys, COMSOL, Altair, etc.

Bibliography

[1] Lars Andersen. *Linear Elastodynamic Analysis*. DCE Lecture Notes 3. Aalborg, Denmark: Department of Civil Engineering, Aalborg University, 2006.

[2] Hans-Peter Wallin, U Carlsson, M Åbom, H Bodén and R Glav. *Sound and vibration*. 2nd ed. Translated by R. Hildebrand. Stockholm: Institutionen för farkostteknik, Tekniska högskolan, 2010. ISBN: 9789174155532.

[3] SIMULIA Abaqus 2016. *1.6 A quick review of the finite element method*. URL: `http://130.149.89.49:2080/v2016/books/gsa/default.htm?startat=ch01s06.html` (visited on 19/03/2025).

[4] Niels Ottosen and Hans Petersson. *Introduction to the finite element method*. Prentice Hall, 1992.

[5] SIMULIA Abaqus 2016. *2. Abaqus Basics*. URL: `http://130.149.89.49:2080/v2016/books/gsa/default.htm?startat=ch02.html` (visited on 19/03/2025).

[6] SIMULIA Abaqus 2016. *2.2 Introduction to Abaqus/CAE*. URL: `http://130.149.89.49:2080/v2016/books/gsa/default.htm?startat=ch02s02.html` (visited on 19/03/2025).

[7] SIMULIA Abaqus 2016. *Abaqus Keywords Reference Guide*. URL: `http://130.149.89.49:2080/v2016/books/key/default.htm` (visited on 19/03/2025).

[8] SIMULIA Abaqus 6.14. *2.2 Format of the input file*. URL: `http://62.108.178.35:2080/v6.14/books/gsk/default.htm?startat=ch02s02.html` (visited on 02/05/2025).

[9] SIMULIA Abaqus 6.14. *27.1.1 Element library: overview*. URL: `http://62.108.178.35:2080/v6.14/books/usb/default.htm?startat=pt06ch27s01abo25.html` (visited on 09/05/2025).

[10] SIMULIA Abaqus 6.14. *28.1 General-purpose continuum elements*. URL: `http://62.108.178.35:2080/v6.14/books/usb/default.htm?startat=pt06ch28s01.html` (visited on 09/05/2025).

[11] SIMULIA Abaqus 6.14. *3.3.1 Solid infinite elements*. URL: `http://62.108.178.35:2080/v6.14/books/stm/default.htm?startat=ch03s03ath68.html` (visited on 06/05/2025).

[12] SIMULIA Abaqus 6.14. *28.3.2 Infinite element library*. URL: `http://62.108.178.35:2080/v6.14/books/usb/default.htm?startat=pt06ch28s03ael08.html` (visited on 06/05/2025).

[13] GeeksforGeeks. *Python OOPs Concepts*. 2025. URL: `https://www.geeksforgeeks.org/python-oops-concepts/` (visited on 07/05/2025).

[14] Python. *The Python Standard Library*. URL: `https://docs.python.org/3/library/index.html` (visited on 07/05/2025).

[15] Christophe Geuzaine. *Gmsh Python tutorial.* URL: `https://gitlab.onelab.info/gmsh/gmsh/-/tree/gmsh_4_13_1/tutorials/python` (visited on 08/05/2025).

[16] Gmsh 4.13.1. *Gmsh tutorial.* URL: `https://gmsh.info/doc/texinfo/gmsh.html#Gmsh-tutorial` (visited on 08/05/2025).

Appendix A

Input file example

The following is the complete input file used for creating the cube in Figure 2.4.

```
*Heading
Ground model
**
*Preprint, echo=NO, model=NO, history=NO, contact=NO
**
** PARTS
**
*Part, name=Ground
*End Part
**
** ASSEMBLY
**
*Assembly, name=GroundAssembly
**
*Instance, name=Ground-1, part=Ground
**
**---------------------------------------------------------------------------
** Model definition
**---------------------------------------------------------------------------
**
** NODES
**
*Node
      1,         0.0,          0.0,          0.0
      2,         1.0,          0.0,          0.0
      3,         1.0,          1.0,          0.0
      4,         0.0,          1.0,          0.0
      5,         0.0,          0.0,          1.0
      6,         1.0,          0.0,          1.0
      7,         1.0,          1.0,          1.0
      8,         0.0,          1.0,          1.0
      9,         0.5,          0.0,          0.0
     10,         1.0,          0.5,          0.0
     11,         0.5,          1.0,          0.0
     12,         0.0,          0.5,          0.0
     13,         0.5,          0.0,          1.0
     14,         1.0,          0.5,          1.0
     15,         0.5,          1.0,          1.0
     16,         0.0,          0.5,          1.0
     17,          0.0,          0.0,          0.5
     18,          1.0,          0.0,          0.5
     19,          1.0,          1.0,          0.5
     20,          0.0,          1.0,          0.5
**
** ELEMENTS
**
*Element, type=C3D20, elset=Set-layer1
1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
   16, 17, 18, 19, 20
**
** ELEMENT SECTION PROPERTIES
**
*Solid Section, elset=Set-layer1, material=Ground-1
,
**
*End Instance
**
*Nset, nset=LoadingPoint, instance=Ground-1
5
*Nset, nset=ObservationPoint, instance=Ground-1
6
**
*End Assembly
**
** MATERIALS
**
*Material, name=Ground-1
*Damping, structural=0.06
*Density
2000.0
*Elastic
160000000.0, 0.33
**
**----------------------------------------------------------------------------
** History Data
**----------------------------------------------------------------------------
**
** STEP: Step-1
**
*Step, name=Step-1, nlgeom=NO, perturbation
*Steady State Dynamics, direct, frequency scale=LINEAR, friction damping=NO
50.0, 51.0, 2, 1
**
** LOADS
** Name: PointLoad Type: Concentrated force
*Cload, real
LoadingPoint, 3, -1
**
** OUTPUT REQUESTS
**
```

```
*Output, field
*Node Output
A, U, V
**
** HISTORY OUTPUT: H-Output-1
**
*Output, history
*Node Output, nset=ObservationPoint
U1, U2, U3, V1, V2, V3, A1, A2, A3
*End Step
**
**----------------------------------------------------------------------
** END OF INPUT FILE
**----------------------------------------------------------------------
```

Appendix B

User manual

The following files are needed and should be placed in the same folder for the program to work:

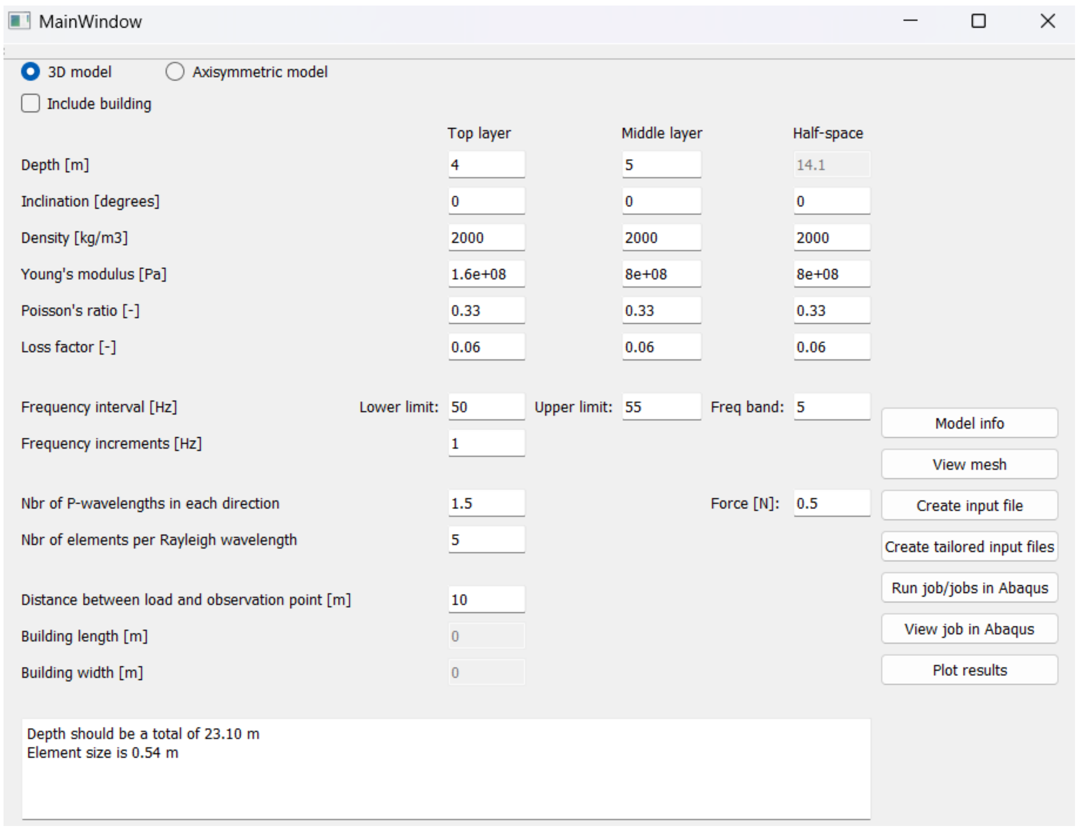
- mainprogram.py
- groundmodel.py
- odbprogram.py
- mainwindow.ui
- building-only-instances.inp
- 1-100Hz-step0.5Hz.txt
- 1-100Hz-step0.5Hz-building.txt

The user interface is started by running `mainprogram.py`. The interface provides two ground model choices, a 3D model or an axisymmetric model. For the 3D model, the user is given the choice of including a building. The default setting when starting the interface is the 3D ground model without a building included, see Figure B.1.

The user can then specify the material properties, depth and inclination of three layers as well as the lower and upper boundaries of the frequency interval, the size of the frequency increments and the size of any frequency bands. Furthermore, the user chooses the number of P-wavelengths that should be used for determining model size and the number of elements per Rayleigh wavelength that should be used for determining element size. Finally, the distance between the load and observation point should be specified as well as the size of the load. If a building is included, the length and width of the building should also be specified.

Note that the user interface lacks error handling and can not handle invalid data types as entered values, nor does it handle unreasonable values of the correct data type. If either is used, it will lead the program to crash.

The depth can only be adjusted by the user for the top and middle layer, the half-space will adjust its depth based on what the total depth should be for the chosen values, which is communicated to the user in the text box along with the current element size based on the chosen values. If the sum of the depths for the top and middle layers exceeds what the total depth should be, the user is warned and the half-space is adjusted to a minimal depth of 0.1 m, see Figure B.2.



[✓ 3D model] Axisymmetric model

Include building

	Top layer	Middle layer	Half-space
Depth [m]	4	5	14.1
Inclination [degrees]	0	0	0
Density [kg/m3]	2000	2000	2000
Young's modulus [Pa]	1.6e+08	8e+08	8e+08
Poisson's ratio [-]	0.33	0.33	0.33
Loss factor [-]	0.06	0.06	0.06

Frequency interval [Hz] Lower limit: 50 Upper limit: 55 Freq band: 5

Frequency increments [Hz] 1

Nbr of P-wavelengths in each direction 1.5 Force [N]: 0.5

Nbr of elements per Rayleigh wavelength 5

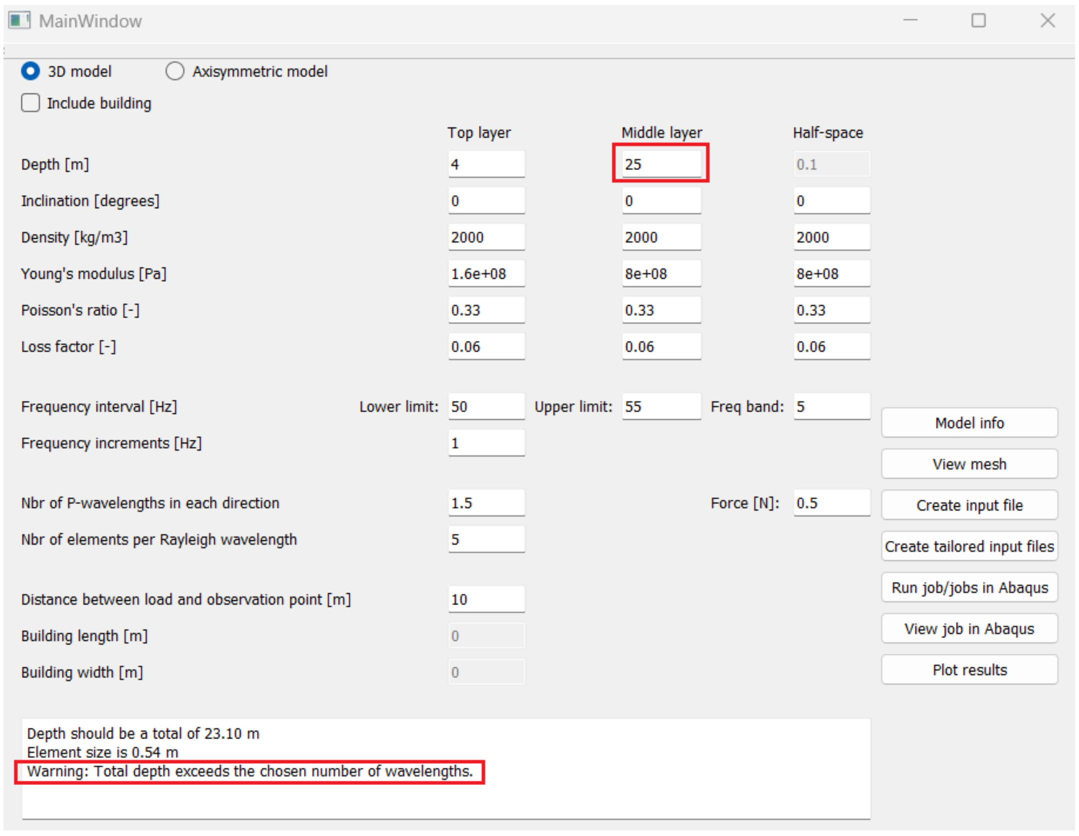
Distance between load and observation point [m] 10

Building length [m] 0

Building width [m] 0

```
Depth should be a total of 23.10 m
Element size is 0.54 m
```

Figure B.1: The default settings of the user interface.



[✓ 3D model] Axisymmetric model

Include building

	Top layer	Middle layer	Half-space
Depth [m]	4	25	0.1
Inclination [degrees]	0	0	0
Density [kg/m3]	2000	2000	2000
Young's modulus [Pa]	1.6e+08	8e+08	8e+08
Poisson's ratio [-]	0.33	0.33	0.33
Loss factor [-]	0.06	0.06	0.06

Frequency interval [Hz] Lower limit: 50 Upper limit: 55 Freq band: 5

Frequency increments [Hz] 1

Nbr of P-wavelengths in each direction 1.5 Force [N]: 0.5

Nbr of elements per Rayleigh wavelength 5

Distance between load and observation point [m] 10

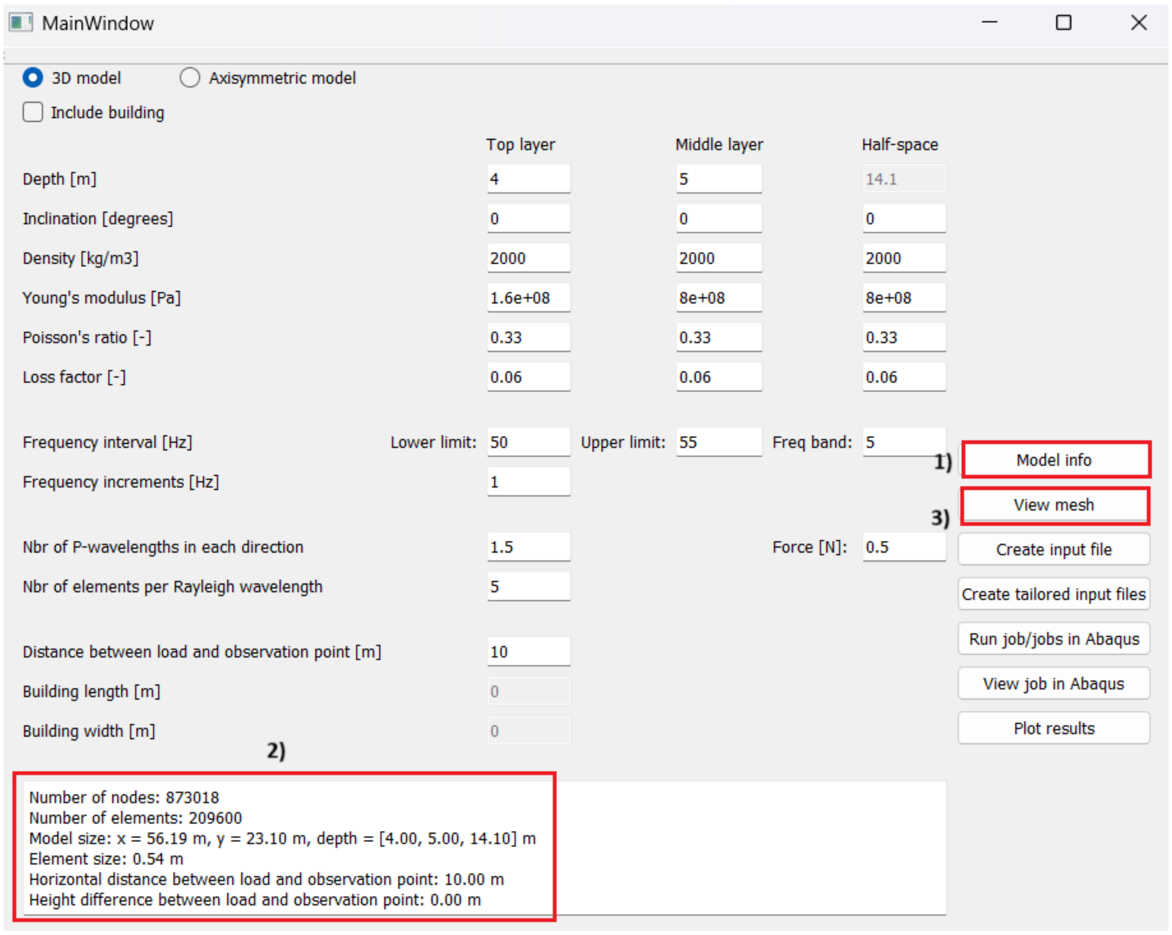
Building length [m] 0

Building width [m] 0

```
Depth should be a total of 23.10 m
Element size is 0.54 m
Warning: Total depth exceeds the chosen number of wavelengths.
```

Figure B.2: The depth of the two top layers exceed the total depth.

The user can choose to show model information, Figure B.3, and to view the created geometry and mesh in Gmsh, Figure B.4.



(●) 3D model ( ) Axisymmetric model
[ ] Include building

	Top layer	Middle layer	Half-space
Depth [m]	4	5	14.1
Inclination [degrees]	0	0	0
Density [kg/m3]	2000	2000	2000
Young's modulus [Pa]	1.6e+08	8e+08	8e+08
Poisson's ratio [-]	0.33	0.33	0.33
Loss factor [-]	0.06	0.06	0.06

Frequency interval [Hz] Lower limit: 50 Upper limit: 55 Freq band: 5
Frequency increments [Hz] 1
Nbr of P-wavelengths in each direction 1.5 Force [N]: 0.5
Nbr of elements per Rayleigh wavelength 5
Distance between load and observation point [m] 10
Building length [m] 0
Building width [m] 0

```
Number of nodes: 873018
Number of elements: 209600
Model size: x = 56.19 m, y = 23.10 m, depth = [4.00, 5.00, 14.10] m
Element size: 0.54 m
Horizontal distance between load and observation point: 10.00 m
Height difference between load and observation point: 0.00 m
```

Figure B.3: Model information for the 3D model with the chosen parameters.

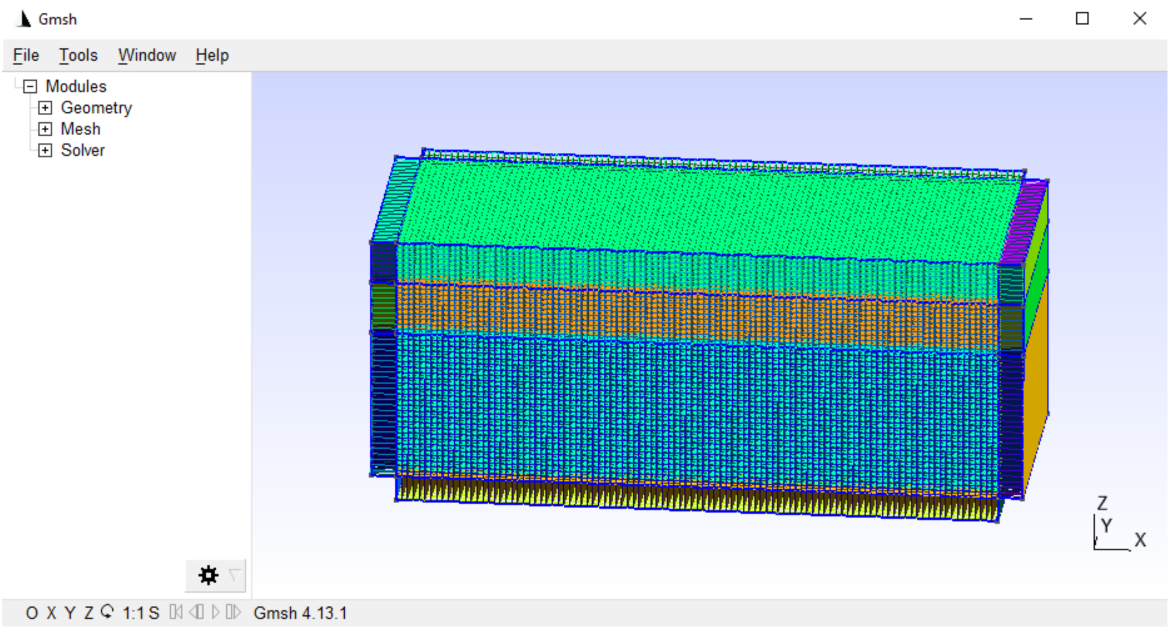
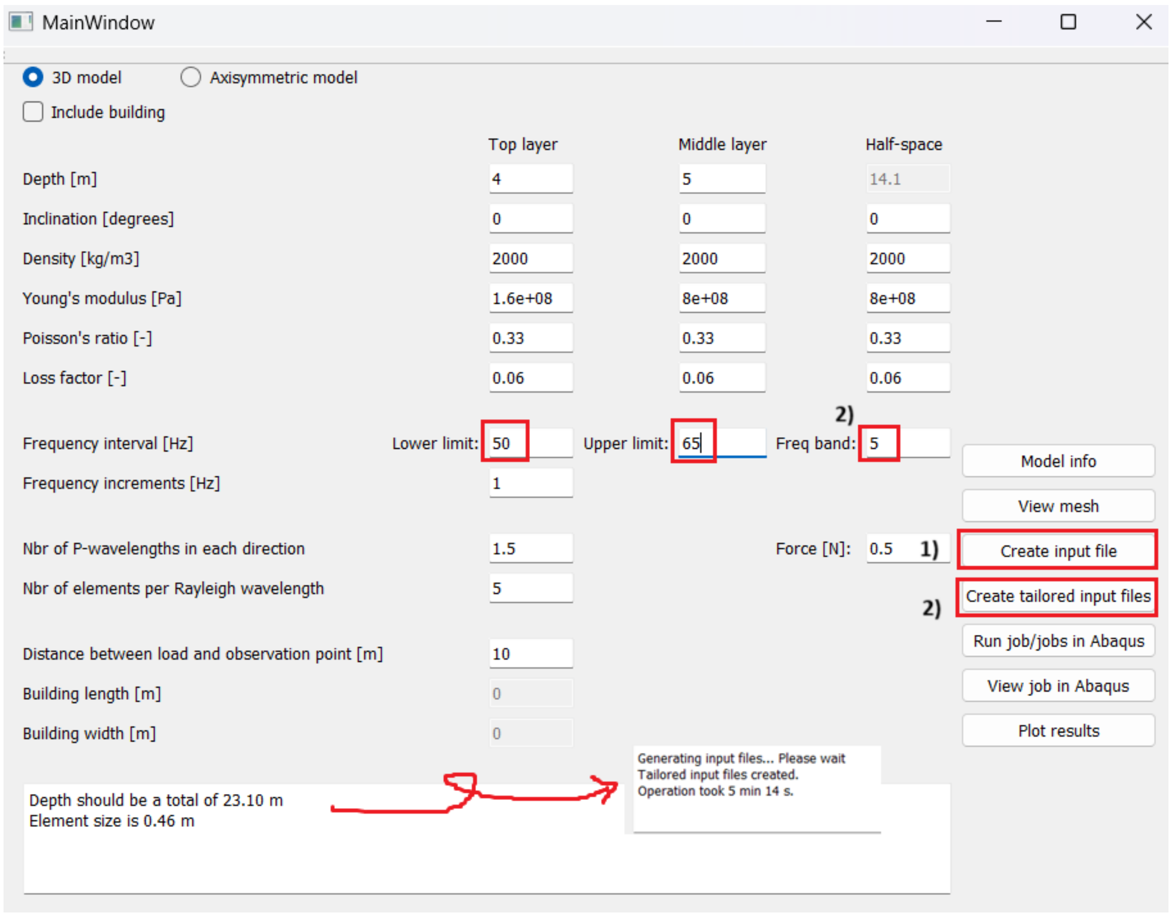


Figure B.4: Visualisation of the 3D model with the chosen parameters.

For a given frequency interval, for example 50-65 Hz, the user is given the choice of creating one input file for the full interval or multiple input files consisting of user-defined sized frequency bands, for example 5 Hz bands. The user is alerted as to the operation time for creating the input file/files. See Figure B.5.



MainWindow

[✓ 3D model] Axisymmetric model

Include building

	Top layer	Middle layer	Half-space
Depth [m]	4	5	14.1
Inclination [degrees]	0	0	0
Density [kg/m3]	2000	2000	2000
Young's modulus [Pa]	1.6e+08	8e+08	8e+08
Poisson's ratio [-]	0.33	0.33	0.33
Loss factor [-]	0.06	0.06	0.06

Frequency interval [Hz] Lower limit: 50 Upper limit: 65 Freq band: 5

Frequency increments [Hz] 1

Nbr of P-wavelengths in each direction 1.5

Force [N]: 0.5

Nbr of elements per Rayleigh wavelength 5

Distance between load and observation point [m] 10

Building length [m] 0

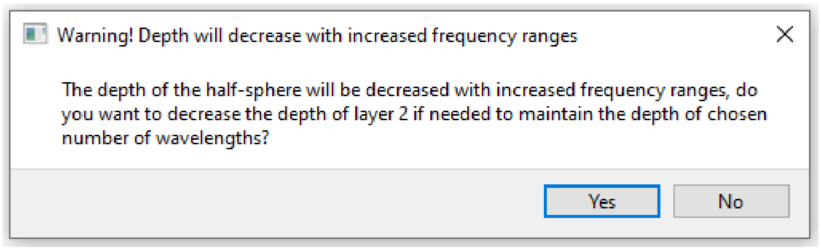
Building width [m] 0

Depth should be a total of 23.10 m
Element size is 0.46 m

Generating input files... Please wait
Tailored input files created.
Operation took 5 min 14 s.

Figure B.5: Generating the input files.

If "Create tailored input files" is chosen, the user gets asked if the depth of the middle layer should decrease if the sum of the depths of the top and middle layer exceed what the total depth should be, Figure B.6. If the user chooses "No", only the half-space will decrease in depth until it reaches the minimum required depth of 0.1 m, after which the total depth is not decreased further.

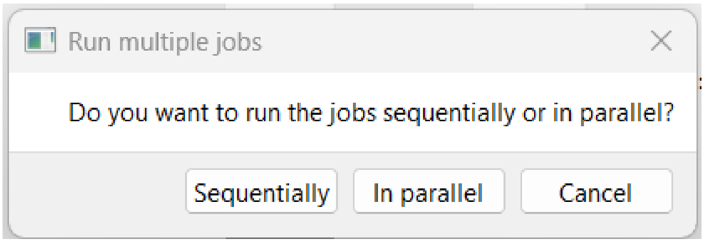


Warning! Depth will decrease with increased frequency ranges

The depth of the half-sphere will be decreased with increased frequency ranges, do you want to decrease the depth of layer 2 if needed to maintain the depth of chosen number of wavelengths?

Figure B.6: Warning of decreasing depth.

The input file/files can then be sent for analyses in Abaqus through "Run job/jobs in Abaqus”. When clicking this button, the user gets to choose one or multiple input files to submit. If more than one input file is chosen, the user gets asked if the jobs should run sequentially or in parallel, Figure B.7. The interface will, through the text box, let the user know when the analyses is completed and the total elapsed time for the analyses. The interface can be used even when analyses are running.

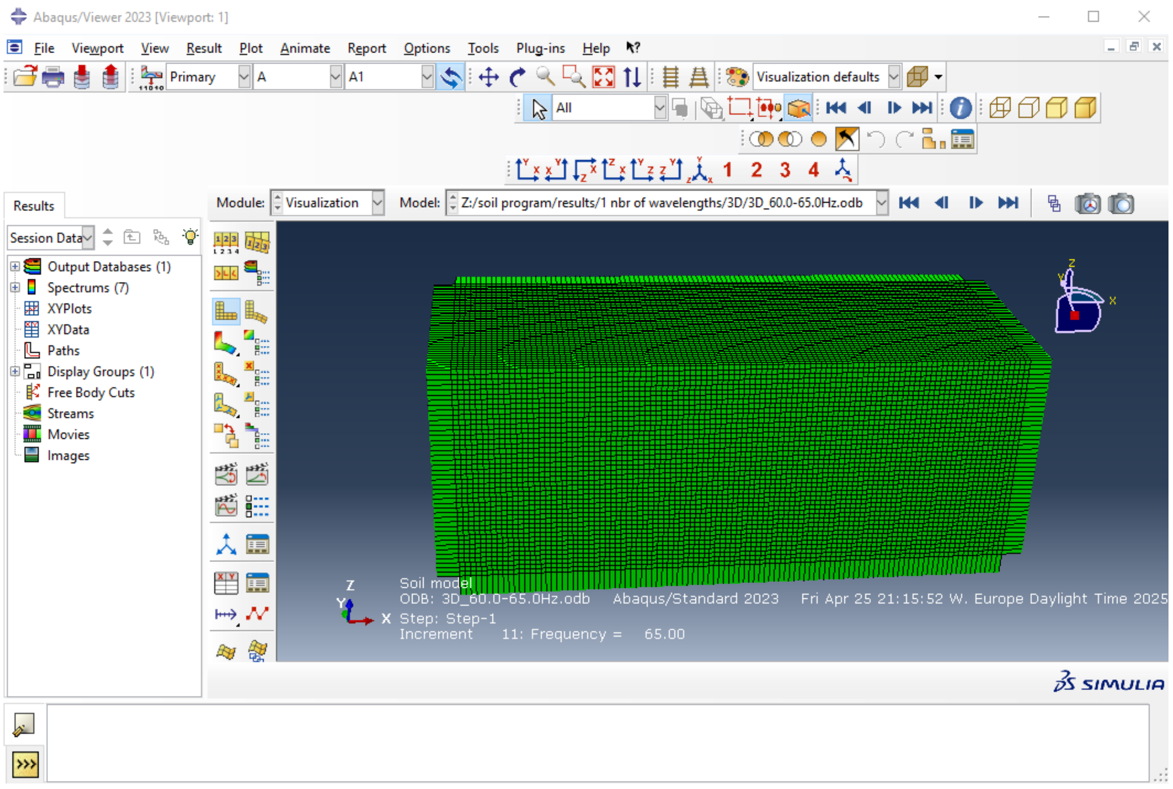


Run multiple jobs

Do you want to run the jobs sequentially or in parallel?

Figure B.7: Sending the jobs for analyses in Abaqus.

If ”View job in Abaqus” is clicked, the user gets to choose an odb-file which launches Abaqus and allows the user to analyse the results in Abaqus/CAE, Figure B.8.



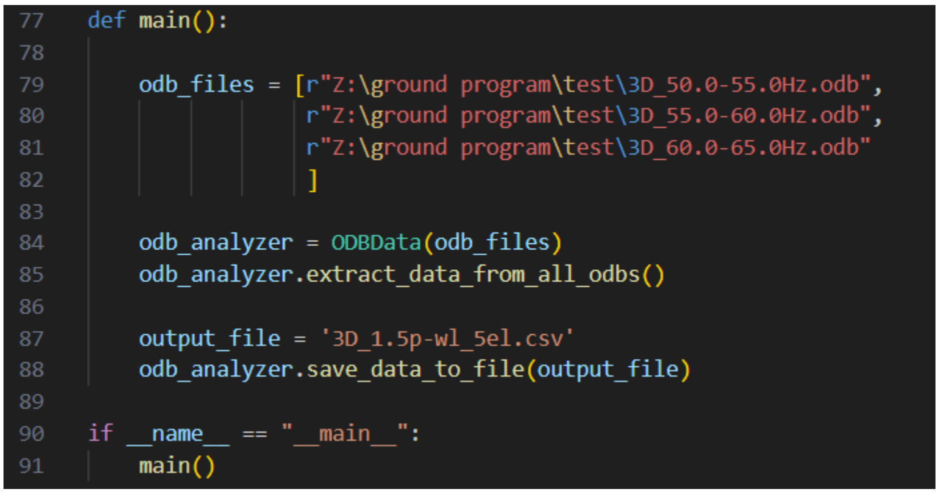
Abaqus/Viewer 2023 [Viewport: 1]

Module: Visualization   Model: Z:/soil program/results/1 nbr of wavelengths/3D/3D_60.0-65.0Hz.odb

```
Soil model
ODB: 3D_60.0-65.0Hz.odb    Abaqus/Standard 2023    Fri Apr 25 21:15:52 W. Europe Daylight Time 2025
Step: Step-1
Increment    11: Frequency =    65.00
```

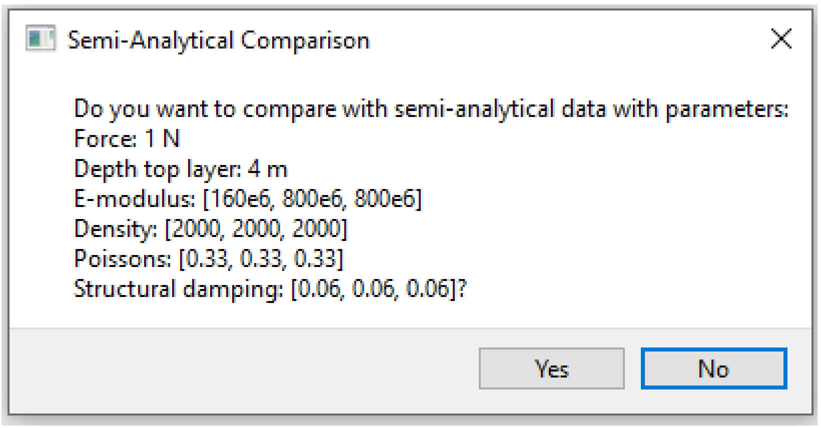
Figure B.8: Viewing the results in Abaqus/CAE.

The results can also be analysed without the Abaqus/CAE. This is done in two steps, first using `odbprogram.py` which extracts displacements, velocities and accelerations from odb-files and saves them into a csv- or text-file, Figure B.9, and then by clicking ”Plot results” in the user interface which gives the user the option of choosing a csv- or text-file and plotting the data from the files. The user also gets to chose if results from semi-analytical data should be plotted for comparison, Figure B.10.



```
def main():

    odb_files = [r"Z:\ground program\test\3D_50.0-55.0Hz.odb",
                 r"Z:\ground program\test\3D_55.0-60.0Hz.odb",
                 r"Z:\ground program\test\3D_60.0-65.0Hz.odb"
                 ]

    odb_analyzer = ODBData(odb_files)
    odb_analyzer.extract_data_from_all_odbs()

    output_file = '3D_1.5p-wl_5el.csv'
    odb_analyzer.save_data_to_file(output_file)

if __name__ == "__main__":
    main()
```

Figure B.9: Creating a csv-file of the extracted data from odb-files.



Semi-Analytical Comparison

Do you want to compare with semi-analytical data with parameters:
Force: 1 N
Depth top layer: 4 m
E-modulus: [160e6, 800e6, 800e6]
Density: [2000, 2000, 2000]
Poissons: [0.33, 0.33, 0.33]
Structural damping: [0.06, 0.06, 0.06]?

Figure B.10: Option of comparing the results from the FE models with the semi-analytical model.