Development of a parallel finite-element tool for dynamic soil-structure interaction

A preliminary case study on the dynamic stiffness of a vertical pile

MÅRTEN ULLBERG

Master of Science Thesis
Stockholm, Sweden 2012
Development of a Parallel Finite-element Tool for Dynamic Soil-structure Interaction

A Preliminary Case Study on the Dynamic Stiffness of a Vertical Pile

Mårten Ullberg

July 2012
TRITA-BKN. Master Thesis 360, 2012
ISSN 1103-4297
ISRN KTH/BKN/EX-360-SE
Summary

This thesis has two major goals; first to develop scalable scripts for steady-state analysis, then to perform a case study on the dynamic properties of a vertical pile. The scripts are based on the numerical library PETSc for parallel linear algebra. This opens up the opportunity to use the scripts to solve large-scale models on supercomputers. The performance of the scripts are verified against problems with analytical solutions and the commercial software ABAQUS. The case study compares the numerical results with those obtained from an approximate solution.

The results from this thesis are verified scripts that can find a steady-state solution for linear-elastic isotropic solids on supercomputers. The case study has shown differences between numerical and semi-analytical solutions for a vertical pile. The dynamic stiffness show differences within reasonable limits but the equivalent viscous damping show larger differences. This is believed to come from the material damping in the soil that has been excluded from the approximate solution.

These two results make it possible for further case studies on typical three-dimensional problems, that result in large-scale models, such as the dynamic properties of a slanted pile or pile-groups. The scripts can easily be expanded and used for other interesting research projects and this is the major outcome of from this thesis.

Keywords: dynamic stiffness and damping, finite-element analysis, steady-state analysis, parallel calculations, PETSc, piles, pile-groups.
Sammanfattning


Dessa två resultat möjliggör vidare fallstudier av typiska problem i tre dimensioner, så som de dynamiska egenskaperna hos lutande pålar och pålgrupper. Skripten kan lätt utvecklas vidare till att användas i nya intressanta forskningsprojekt och detta får ses som den stora utkomsten av denna uppsats.

Nyckelord: dynamisk styvhet och dämpning, finita-elementanalys, fortvarighetstillstånd, parallelliserade beräkningar, PETSc, pålar, pålgrupper.
Preface

The research presented in this report is a master thesis at the Division of Structural Engineering and Bridges, The Royal Institute of Technology, KTH. The work was initiated and carried out at Tyréns AB in Stockholm.

First, I would like to thank my supervisor Tech. Lic Mahir Ülker-Kaustell, Ph.D student at the Department of Civil and Architectural Engineering at the Royal Institute of Technology, for his great support and enthusiasm for my work with this thesis. Without him, this thesis would not have been possible. I would like to thank my examiner Prof. Raid Karoumi for his support and guidance. I also would like to thank both of them for their careful proofreading.

I would like to thank the bridge department at Tyréns AB for making it possible for me to carry out my work at their department.

Stockholm, June 2012

Mårten Ullberg
Contents

Summary i
Sammanfattning iii
Preface v
List of Abbreviations xi

1 Introduction 1
  1.1 Problem Description ........................................... 1
  1.2 Aim and Scope .................................................. 2
  1.3 Thesis Structure ............................................... 2

2 Theoretical Background 3
  2.1 Structural Dynamics .......................................... 3
    2.1.1 Linear Theory of Elasticity .............................. 3
    2.1.2 Soil-structure Interaction ............................... 4
    2.1.3 Dynamic Stiffness Function ............................. 5
    2.1.4 Damping .................................................. 5
  2.2 Dynamics of Piles and Pile Groups ......................... 6
    2.2.1 Dynamics of Piles ....................................... 6
    2.2.2 Dynamics of Pile Groups ............................... 7
    2.2.3 Stiffness and Damping Constants for Vertical Piles .... 8
  2.3 The Finite-element Method ................................ 11
    2.3.1 Element Matrices ....................................... 11
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.2 Surface Traction</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3 Steady-state Response Spectrum</td>
<td>12</td>
</tr>
<tr>
<td>2.3.4 Radiating Boundary</td>
<td>13</td>
</tr>
<tr>
<td>2.3.5 Wave Propagation in Elastic Medium</td>
<td>14</td>
</tr>
<tr>
<td>2.4 Parallel FEA</td>
<td>15</td>
</tr>
<tr>
<td>2.4.1 Object-oriented Programming</td>
<td>16</td>
</tr>
<tr>
<td>2.4.2 MPI</td>
<td>16</td>
</tr>
<tr>
<td>2.4.3 PETSc</td>
<td>17</td>
</tr>
<tr>
<td>2.4.4 SLEPc</td>
<td>19</td>
</tr>
<tr>
<td>2.4.5 MUMPS</td>
<td>19</td>
</tr>
<tr>
<td>2.4.6 Distributed SuperLU</td>
<td>20</td>
</tr>
<tr>
<td>2.5 Iterative Methods for Solving Linear Systems</td>
<td>21</td>
</tr>
<tr>
<td>2.5.1 Background in Relevant Linear Algebra</td>
<td>21</td>
</tr>
<tr>
<td>2.5.2 Krylov Subspace Methods</td>
<td>24</td>
</tr>
<tr>
<td>2.5.3 Preconditioners</td>
<td>25</td>
</tr>
<tr>
<td>2.5.4 Parameter-Dependent System of Equations</td>
<td>27</td>
</tr>
<tr>
<td>3 Scripts</td>
<td>29</td>
</tr>
<tr>
<td>3.1 General Philosophy</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1 Input File Properties</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Objects</td>
<td>31</td>
</tr>
<tr>
<td>3.3 PETSc Functions</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1 Matrix and Vector Assembly</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2 Steady-state Solution Procedure</td>
<td>34</td>
</tr>
<tr>
<td>4 Verification</td>
<td>37</td>
</tr>
<tr>
<td>4.1 Elementary Problems of Elasticity in Three Dimensions</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1 Stretching of a Prismatical Bar by its Own Weight</td>
<td>37</td>
</tr>
<tr>
<td>4.1.2 Pure Bending of Prismatical Bars</td>
<td>38</td>
</tr>
<tr>
<td>4.2 Natural Frequencies and Mode Shapes</td>
<td>39</td>
</tr>
</tbody>
</table>
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate Gradient Method</td>
</tr>
<tr>
<td>CSR</td>
<td>Compressed Sparse Row</td>
</tr>
<tr>
<td>C3D4</td>
<td>Solid 4-node linear tetrahedron element</td>
</tr>
<tr>
<td>C3D10</td>
<td>Solid 10-node quadratic tetrahedron element</td>
</tr>
<tr>
<td>C3D8</td>
<td>Solid 8-node linear hexahedron element</td>
</tr>
<tr>
<td>C3D20</td>
<td>Solid 20-node quadratic hexahedron element</td>
</tr>
<tr>
<td>EPS</td>
<td>Eigenvalue Problem Solver</td>
</tr>
<tr>
<td>FEA</td>
<td>Finite-element Analysis</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite-element Method</td>
</tr>
<tr>
<td>GMRES</td>
<td>Generalized Minimal Residual Method</td>
</tr>
<tr>
<td>ILU</td>
<td>Incomplete LU factorization</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MUMPS</td>
<td>Multifrontal Massively Parallel Solver</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-oriented Programming</td>
</tr>
<tr>
<td>PETSc</td>
<td>Portable, Extensive Toolkit for Scientific Computation</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SLEPc</td>
<td>Scalable Library for Eigenvalue Problem Computations</td>
</tr>
<tr>
<td>SS</td>
<td>Steady-state</td>
</tr>
<tr>
<td>SSI</td>
<td>Soil-structure interaction</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Problem Description

A dynamic evaluation should be performed for train bridges that are designed for speeds exceeding 200 km/h, according to the requirements given by the Swedish Transport Administration (Trafikverket, 2011). In Andersson et al. (2010), a number of elements are discussed when dealing with the dynamic analysis of a bridge. Dynamic evaluation is usually performed numerically with a commercial tool for finite-element analysis (FEA). It is pointed out that it is of outmost importance that the numerical model should be able to act out the real dynamic behavior of the bridge. Among other things, they identify boundary conditions as one important factor when modeling a bridge and how soil-structure interaction (SSI) is a subject where further research is necessary.

Ülker-Kaustell (2009) showed how SSI could influence the dynamic behavior of a bridge. In this research, it is for instance described that how bridge deck accelerations were found for a bridge on a piled foundation. Using boundary conditions that neglect SSI could lead to non-conservative assumptions during dynamic analysis. SSI can be modeled as frequency depending springs and dampers. Different alternatives exist when establishing the frequency dependent boundary condition; analytical, semi-analytical or functions obtained from numerical analysis. The analytical or semi-analytical solutions are limited to simple geometries of vertical piles. Due to this, numerical analysis could be one way of taking the complex geometry of a pile group into account.

Modeling wave propagation phenomena leads to high demands on both computer hardware and software. Ülker-Kaustell (2009) mentions the possibility to use super-computers for this type of numerical analysis. Given this background, this thesis will address the possibility of developing a scalable tool that can be used on clusters of workstations or supercomputers. The vast amount of fast random access memory (RAM) in these type of systems has great potential when it comes to FEA. Developing stand-alone FEA-scripts with a simple interface will also open up the possibility of extending the scripts into any research subject of interest.
1.2 Aim and Scope

The aim of this master thesis is to develop a tool that makes it possible to investigate the dynamic stiffness of bridge foundations. The tool should be able to perform large-scale finite-element analysis by a direct approach in the frequency domain. The analysis should give complex displacements of a homogeneous linearly elastic solid model with a combination of materials. A radiating boundary should be applied to the edge of the computational domain so that elastic wave reflections are minimized. The development of a new tool for FEA is a huge task and because of the limitations of this thesis, the scripts developed has a problem oriented approach but they also form a good basis for expansion.

The three main goals with this thesis can be summarized as:

- Develop a scalable finite-element tool for linear dynamic soil-structure interaction.
- Verify the tool against problems solved analytically or with commercial finite-element software.
- Compare the dynamic stiffness of a vertical pile with an approximate analytical solution.

1.3 Thesis Structure

First, the theoretical background with literature references necessary to understand the basis of the thesis is described in chapter 2. Chapter 3 describe the scripts that have been developed and chapter 4 their verification. Chapter 5 introduce and explain how the case study is undertaken. Results and discussions from the case study is also presented. Chapter 6 describe conclusions and also suggestions for further research. Appendix A include mode shapes from the verification chapter and appendix B the source code for the scripts.
Chapter 2

Theoretical Background

In this chapter, the necessary theoretical background used in this thesis is presented. Section 2.1 describes a general overview of structural dynamics, section 2.2 dynamics of piles and pile groups and section 2.3 the finite element method. Then, section 2.4, describes the theory and tools used in parallel FEA. Followed by section 2.5 where theoretical background in linear algebra and iterative methods is described. The last section also includes a review of the literature on parameter-dependent systems of equations.

2.1 Structural Dynamics

In contrast to the static structural mechanical problem, which can be idealized through a certain number of unknown parameters, such as material non-linearity or the geometry of the problem, the dynamic aspect of structural engineering introduce new uncertainties such as time-dependent loading and mass acceleration. The goal of structural dynamics is to investigate these time-dependent factors and by doing that, analytically or numerically predict structural behavior. A time-dependent analysis require more than a force equilibrium as for the static analysis. The same stiffness matrix can be used as in the static analysis, but mass and damping matrices need to be computed. This dynamic formulation leads to more advanced mathematical problems also with complex numerical numbers when a direct approach in the frequency domain is used.

The following sections will address some of the general aspects in structural dynamics that are of importance for the case study undertaken in this thesis.

2.1.1 Linear Theory of Elasticity

When a elastic material is undergoing small strain, typically $\varepsilon < 10^{-4}$, the material is in a purely linearly elastic state where the complete deformation will return to its original state upon unloading of the material. This happens to be the case when it
comes to soil vibrations, as described by (Ishihara, 1996, p 1-5). The relationship between stress, $\sigma$, and strain, $\varepsilon$, for such a material is usually described by Hook’s law,

$$\sigma = E\varepsilon,$$

(2.1)

where $E$ is the elastic modulus of the material. Another type of modulus that is related to the elastic modulus is the shear modulus, $G$, that describe the relationship between shear stress, $\tau$, and shear strain, $\gamma$, as

$$\tau = G\gamma,$$

(2.2)

In a three-dimensional space, a material can deform in all three principal directions. A relationship between strains in different directions is Poisson’s ratio. It can be calculated as

$$\nu = -\frac{\varepsilon_{\text{lateral}}}{\varepsilon_{\text{axial}}},$$

(2.3)

and by combining all knowledge about stress and strain, a three-dimensional stress-strain relationship can be derived for a homogeneous, isotropic, linear-elastic material to

$$\begin{bmatrix}
\sigma_{xx} \\
\sigma_{yy} \\
\sigma_{zz} \\
\tau_{xy} \\
\tau_{yz} \\
\tau_{zx}
\end{bmatrix} =
\begin{bmatrix}
(1 - \nu)\hat{E} & \nu\hat{E} & \nu\hat{E} & 0 & 0 & 0 \\
\nu\hat{E} & (1 - \nu)\hat{E} & \nu\hat{E} & 0 & 0 & 0 \\
\nu\hat{E} & \nu\hat{E} & (1 - \nu)\hat{E} & 0 & 0 & 0 \\
0 & 0 & 0 & G & 0 & 0 \\
0 & 0 & 0 & 0 & G & 0 \\
0 & 0 & 0 & 0 & 0 & G
\end{bmatrix}
\begin{bmatrix}
\varepsilon_{xx} \\
\varepsilon_{yy} \\
\varepsilon_{zz} \\
\gamma_{xy} \\
\gamma_{yz} \\
\gamma_{zx}
\end{bmatrix},$$

(2.4a)

where

$$\hat{E} = \frac{E}{(1 - \nu)(1 - 2\nu)}, \quad G = \frac{E}{2(1 + \nu)}.$$

(2.4b)

### 2.1.2 Soil-structure Interaction

How a structure interacts with its surroundings is an interesting topic and very important during dynamic analysis of bridges. The dynamic response of a bridge can be highly dependent on what boundary conditions are used during the analysis. If the interaction with the surroundings is taken into account, a number of uncertainties arise. For instance how well is the foundation slab connected to the underlying gravel, how many layers of soil is necessary to take into account and what are their properties. Another factor is due to plastic deformations around piles, the upper part of the pile may be free from soil which may affect the dynamic properties of the foundation.

The usual design procedure is either to see the bridge rigidly or by springs connected to the foundation slab. An other way to take into account the dynamic stiffness and damping of the foundation is to use a dynamic stiffness function, see section 2.1.3, represented with both elastic springs and viscous dashpots. (Ülker-Kaustell, 2009)
2.1.3 Dynamic Stiffness Function

As described previously, the foundation of a structure can have frequency dependent stiffness and damping properties. This foundation system where interaction between the structure and soil is taking place can be described by a dynamic stiffness function. It describes the variation of stiffness and damping depending on the frequency of the applied load. Through a steady-state analysis, where an harmonic excitation is assumed, the variation of stiffness and damping can be calculated for each excitation frequency. The result from the analysis will be complex and can with a compliance function describe the stiffness and damping of the foundation system. The compliance function can according to Kobori et al. (1971) be calculated as

\[ C = f_1 + i f_2 = \frac{1}{k + i a_0 c}, \]  

where \( a_0 \) is the dimensionless frequency according to equation 2.10, \( k \) is the stiffness

\[ k = \frac{f_1}{f_1^2 + f_2^2}, \]  

and \( c_{eq} \) is the equivalent viscous damping that represent the amount of energy that attenuate due to internal dissipation and wave radiation

\[ c_{eq} = -\frac{f_2}{a_0(f_1^2 + f_2^2)}. \]

2.1.4 Damping

One important aspect in a dynamic analysis is the rate at which the dynamic excitations of a system is decreasing. This is usually referred to as damping. A number of different approaches to damping are presented in the literature. Damping in structures can come from, for instance, the chosen material or how structural members are connected. Two different methods for representing damping in numerical calculations is used in this thesis and will be described in more detail.

Viscous Damping

A mechanical approach to damping is can be found by looking at the ordinary force equilibrium of a single degree-of-freedom system. Here, the force can be related to a velocity through a viscous damping coefficient, \( c \), as

\[ F = c \dot{u}, \]  

where \( F \) is the force and \( \dot{u} \) is the velocity. (Chopra, 2007, p. 13)
CHAPTER 2. THEORETICAL BACKGROUND

Rate-independent Damping

Experiments have shown that energy dissipates internally in materials through strain cycles that are independent of the rate at which the external load is applied (Chopra, 2007, p. 105). But there is no available physical way of describing this rate-independent damping. Ishihara (1996) has proposed a modified non-viscous type Kelvin model where the rate-independence is modeled with a special dashpot. It has been shown that this model can reflect the actual behavior of soil to a sufficient degree. The modified method describes the relationship between shear stress and shear strain as

\[ \tau = (1 + i\eta)G\gamma, \]  

where \( \eta \) is the loss factor that describes the relationship between loss modulus and shear modulus. This damping is often referred to as structural damping and \( \eta = 2\xi \), where \( \xi \) is the damping ratio (Chopra, 2007, p. 107).

2.2 Dynamics of Piles and Pile Groups

The following sections will describe some of the current theories and aspects of dynamic properties of piles and pile groups. Also formulas for calculating the dynamic stiffness and equivalent viscous damping for vertical piles according to the approximate solution from Novak (1974) are presented.

2.2.1 Dynamics of Piles

In 1974, Milos Novak identified that if the dynamic stiffness and damping of soil-pile interaction can be determined, the dynamic response of structures supported by piles could be predicted. He proposed an approximate analytical solution based on linear elasticity. Closed form formulas where derived for calculation of vertical-, horizontal-, rotational- and cross stiffness and damping for a single pile and combination formulas for pile-groups. He found that the dynamic stiffness and damping of piles where dependent on the following dimensionless parameters:

- mass ratio - specified mass of the soil over specified mass of the pile: \( \rho_s/\rho_p \),
- wave velocity ratio - shear wave velocity in the soil over longitudinal wave velocity in the pile: \( V_s/v_c \),
- slenderness ratio - length of pile over pile radius: \( l/r_0 \),
- load ratio - static load over Euler’s buckling load, and
- the dimensionless frequency, as calculated in equation 2.10.
The two most important parameters where the wave velocity and the slenderness ratio. He concluded that both the stiffness and damping of piles would increase with increased wave velocity ratio. But also, that from a practical point of view, the damping can be assumed to be frequency independent. This is due to the fact that usually, the frequencies of interest are $a_0 < 0.5$, where $a_0$ is given by equation 2.10. A numerical study of the difference between a pinned or fixed end of the pile was also performed by Novak. He found that the dynamic stiffness and damping coefficients, equation 2.9, are almost the same for a high pile slenderness ratio. He also compare the formulas to more rigorous numerical calculations of modal analysis. (Novak, 1974)

The closed form formulas presented for dynamic stiffness and damping of piles where later validated against field experiments by Novak and Grigg (1976). Test where performed on single piles and small pile groups. The shear modulus of the soil was found to be the most important parameter for predicting the natural frequencies and resonant amplitudes.

Mylonakis (2001) showed that using a Winkler modulus that is depending on depth, can give accurate and stable predictions of pile behavior for a variety of parameters. He point out that when numerical solutions are used, long piles can give rise to numerical instabilities. The effect on the dynamic properties of piles due to soil heterogeneity was studied by Mylonakis et al. (2005). They conclude that heterogeneity of the soil can arise in the vertical direction from soil sedimentation and in the horizontal direction from when the pile is installed. They show that ignoring the heterogeneity can underestimate stiffness and overestimate damping.

### 2.2.2 Dynamics of Pile Groups

The dynamic behavior of pile groups is much more complicated then that of single piles. A study on the vertical vibration of pile groups was performed by Novak and Sheta (1982), where an approximate theory was presented. They concluded that

- dynamic interaction of piles in a group,
- weakening of the soil around piles due to high strains,
- soil layering and
- pile tip conditions

where some of the important effects that made the dynamic behavior of pile groups very complex. They concluded that because of these factors, the frequency dependence of pile groups are much larger than that of single piles. Later, Kaynia and Kausel (1991) presented a simplified method for evaluating the dynamic response of piles and pile groups in a layered half-space. Their analysis showed that the pile groups behavior is highly frequency dependent and the reason for this primarily because of the pile-soil-pile interaction taking place. Their studies showed also that a
vertically heterogeneous soil, compared to a homogeneous soil, give rise to a higher frequency dependency of stiffness and damping for a pile group.

A simple method for the analysis of pile groups has been presented by Cairo et al. (2005). They identified that for a large pile group, the computational effort is great if numerical methods are used. The simple method that they present for analysis for vertical loaded pile groups is based on the existing theoretical solutions that are available and they verify their method by predicting experimental data, already available in the literature.

Due to the complexity in analysis of pile groups, many of the available theories are based on several assumptions. For instance, the methods derived by Cairo et al. (2005) are based on assumptions of small-strains and soil linearity, perfect bonding between soil and pile, free-field soil displacements to simulate pile-soil-pile interaction. An other aspect is also that the piles are usually vertical with a fixed spacing.

Later, expression for approximating the horizontal dynamic stiffness of pile groups were derived by Dai and Roesset (2010). They propose a simple method for preliminary estimates assuming linear material properties. Formulas for modeling the whole foundation system, pile group and soil, as a single-degree of freedom system were presented.

2.2.3 Stiffness and Damping Constants for Vertical Piles

The following section is a summary of the results presented by Novak (1974). The approximated equivalent stiffness and damping frequency dependent functions for vertical, $zz$, horizontal, $xx$, cross, $x\psi$, and rotational, $\psi\psi$ can, by the simplified expression, be estimated with

$$k_{zz} = \frac{E_p A}{l} F_{18}(\Lambda), \quad c_{zz} = \frac{E_p A}{\omega l} F_{18}(\Lambda)_2,$$

$$k_{xx} = \frac{E_p I}{l^3} F_{11}(\lambda), \quad c_{xx} = \frac{E_p I}{\omega l} F_{11}(\lambda)_2,$$

$$k_{x\psi} = \frac{E_p I}{l^2} F_9(\lambda), \quad c_{x\psi} = \frac{E_p I}{\omega l^2} F_9(\lambda)_2,$$

$$k_{\psi\psi} = \frac{E_p I}{l} F_7(\lambda), \quad c_{\psi\psi} = \frac{E_p I}{\omega l} F_7(\lambda)_2,$$

where $E_p$, $A$, $I$, $l$, $\omega$ is the Young’s modulus, cross-section area, moment of inertia, length and circular frequency, respectively. For a pile with a fixed lower end, stiffness and damping is found through substitution of $F_7$ with $F_2$, $F_9$ with $F_4$ and $F_{11}$ with $F_6$. The dimensionless frequency, $a_0$, used by Novak is calculated as

$$a_0 = r_0 \omega \sqrt{\rho/G},$$

where $r_0$ and $\rho$ is the equivalent radius and mass density of the pile. $G$ is the shear modulus of the soil. Parameter $F_i$ is the frequency function where $F_i(\cdot)_1$ and $F_i(\cdot)_2$
is the real and imaginary parts of the complex frequency function, respectively. The formulas for frequency functions are plotted by Novak (1974) for a typical set of dimensionless parameters \((l/r_0 = 40, V_s/v_c = 0.03, \rho_s/\rho_p = 0.7, \nu = 0.4)\). The figure presented by Novak has been reproduced so that the replicated formulas can be validated, see figure 2.1. The formulas are highly dependent on the input data and an exact replication is very hard to find because the input data used by Novak has not been explicitly given. The data for replicating the figures is therefore estimates and results are only approximate. But still, in a sufficiently correct order of magnitude and also the behavior of the functions are presumably correct.

**Frequency function for horizontal translation and rotation**

The frequency function for a pile with the lower end fixed, \(F_2, F_4, F_6\), and pinned, \(F_7, F_9, F_{11}\), is calculated as

\[
F_2(\lambda) = -\lambda \left[ \frac{\cosh \lambda \sin \lambda - \sinh \lambda \cos \lambda}{\cosh \lambda \cos \lambda - 1} \right], \quad (2.11a)
\]

\[
F_4(\lambda) = \lambda^2 \left[ \frac{\sinh \lambda \sin \lambda}{\cosh \lambda \cos \lambda - 1} \right], \quad (2.11b)
\]

\[
F_6(\lambda) = -\lambda^3 \left[ \frac{\cosh \lambda \sin \lambda + \sinh \lambda \cos \lambda}{\cosh \lambda \cos \lambda - 1} \right], \quad (2.11c)
\]

\[
F_7(\lambda) = \lambda \left[ \frac{2 \sinh \lambda \sin \lambda}{\cosh \lambda \sin \lambda - \sinh \lambda \cos \lambda} \right], \quad (2.11d)
\]

\[
F_9(\lambda) = -\lambda^2 \left[ \frac{\cosh \lambda \sin \lambda + \sinh \lambda \cos \lambda}{\cosh \lambda \sin \lambda - \sinh \lambda \cos \lambda} \right], \quad (2.11e)
\]

\[
F_{11}(\lambda) = \lambda^3 \left[ \frac{2 \cosh \lambda \cos \lambda}{\cosh \lambda \sin \lambda - \sinh \lambda \cos \lambda} \right], \quad (2.11f)
\]

and the complex frequency parameter for horizontal translation and rotation is

\[
\lambda = l \left[ \frac{1}{E_p I} \left[ \mu \omega^2 - GS_{u1} - i(\omega + GS_{u2}) \right] \right]^{1/4}, \quad (2.12)
\]

where \(\mu\) is the mass of pile per unit length and coefficient \(c\) is the internal damping of a pile. \(S_{u1}\) and \(S_{u2}\) is the real and complex part of equation 2.15a.

**Frequency function for vertical translation**

The frequency function for vertical translation is

\[
F_{18}(\Lambda) = \Lambda \cot \Lambda \quad (2.13)
\]

and the complex frequency parameter for vertical translation is calculated as

\[
\Lambda = l \left[ \frac{1}{E_p A} \left[ \mu \omega^2 - GS_{w1} - i(\omega + GS_{w2}) \right] \right]^{1/2}, \quad (2.14)
\]

where \(\mu\) is the mass of pile per unit length and coefficient \(c\) is the internal damping of a pile. \(S_{w1}\) and \(S_{w2}\) is calculated by equation 2.15b and 2.15c, respectively.
Layer reaction parameters

Layer reaction parameters has been reproduced from Novak (1974), see figure 2.2, by

\[
S_w(a_0, \nu) = 2\pi a_0 \frac{1}{\sqrt{q}} H_2^{(2)}(a_0) H_1^{(2)}(x_0) + H_2^{(2)}(x_0) H_1^{(2)}(a_0),
\]

(2.15a)

\[
S_{w1}(a_0) = 2\pi a_0 \frac{J_1(a_0) J_0(a_0) + Y_1(a_0) Y_0(a_0)}{J_0^2(a_0) + Y_0^2(a_0)},
\]

(2.15b)

\[
S_{w2}(a_0) = \frac{4}{J_0^2(a_0) + Y_0^2(a_0)},
\]

(2.15c)

where \(J_n, Y_n, H_n^2\) is the Bessel function of first, second and third kind of order \(n\).

\[ q = \frac{1 - 2\nu}{2(1 - \nu)}, \]

\[ x_0 = a_0 \sqrt{q}. \]

Flaws in Equations

The following flaws has been found in Novak (1974):

- In equation [2], the third Hankel function in the numerator should be of order 2 rather then 1, \(H_2^{(2)}(x_0)\).
- In equation [2], \(x_0\) should be dependent on \(q\) as \(x_0 = a_0 \sqrt{q}\)
- In equation [8], the fourth root should be used rather then the square, \(\sqrt[4]{(\cdot)}\)

![Figure 2.1: Validation of FIG.3 as presented by Novak (1974) where frequency functions are plotted for a typical set of dimensionless parameters.](image)
2.3 The Finite-element Method

The finite-element method, FEM, is a mathematical method of analysis where the object of interest is discretized into a finite number of elements. Each element is assigned certain properties corresponding to the actual material. Elements are connected through nodes who are assigned a freedom to translate or rotate in a given number of ways. By doing this, equilibrium equations for each degree of freedom can be formulated by simple means. A static problem can be described by the global equilibrium equation

\[ Ku = F, \]  

(2.16)

where \( K \) is the global stiffness matrix, \( u \) is the displacement vector and \( F \) is the applied force vector. Each row correspond to a degree of freedom of the global system. By solving the linear system of equations, the displacements can be found for a given discretization and applied force. The discretization allows for numerical analysis of complex problems where an analytical solution is impossible to find. (Cook et al., 2001)

2.3.1 Element Matrices

The global stiffness and mass matrix is assembled through submatrices corresponding to each discretized element. An element stiffness matrix is integrated through

\[ K^e = \int \int \int_V \mathbf{B}^T \mathbf{B} \, dV, \]  

(2.17)
CHAPTER 2. THEORETICAL BACKGROUND

where $B$ is the strain-displacement matrix and $D$ is the constitutive relationship for the element material. A consistent element mass matrix, that distribute the mass of the finite-element with the same shape-functions as used for the stiffness, is integrated through

$$
M^e = \iiint_N N^T \rho N \, dV,
$$

(2.18)

where $N$ is the shape-function matrix describing the element geometry and $\rho$ is the density of the element material. Each element matrix is evaluated through full or reduced Gauss-quadrature integration over the element volume, $V$. (Cook et al., 2001)

2.3.2 Surface Traction

Load can be considered in many different ways. One type of load used in FEM is the surface traction type of loading. It consists of applying a second order stress tensor on the element face, by its normal, and then distributing it onto the degrees of freedom of the face nodes. The node forces are calculated through

$$
F^e = \iint_A N^T \sigma N \, dA,
$$

(2.19)

where $\sigma$ is the second-order stress tensor as

$$
\sigma = \begin{bmatrix}
\sigma_{xx} & \tau_{xy} & \tau_{xz} \\
\tau_{yx} & \sigma_{yy} & \tau_{yz} \\
\tau_{zx} & \tau_{zy} & \sigma_{zz}
\end{bmatrix}
$$

(2.20)

and then integrated over the face area, $A$. (Cook et al., 2001)

2.3.3 Steady-state Response Spectrum

A response spectrum is used as a summary of peak responses of a particular excitation motion for a certain system. For each frequency of that applied motion, the investigated quantity in the system is calculated by assuming a full dynamic equilibrium at each frequency. The dynamic equation of motion for a steady-state force can be expressed by the standard finite element discretized equation of motion

$$
M\ddot{u} + C\dot{u} + Ku = F(t),
$$

(2.21)

where $M$, $C$ and $K$ is the mass, damping and stiffness matrix, respectively. $\ddot{u}$, $\dot{u}$ and $u$ is the acceleration, velocity and displacement of each degree of freedom. $F(t)$ is the time dependent externally applied load. The equation of motion has in the frequency domain the following solution for a steady-state harmonic force excitation

$$
[-\omega^2 M + i\omega C + K] \, u = F
$$

(2.22)
and is derived by assuming in equation 2.21 that both the force and displacements are harmonic as

\[ F(t) = F e^{i\omega t}, \quad (2.23a) \]

\[ u(t) = u(\omega) e^{i\omega t} \quad (2.23b) \]

and also with the derivatives to equation 2.23b. The damping matrix \( C = C_r + C_m \) is partly the radiating boundary and the non-viscous material damping. As described in section 2.1.4 the rate-independent material damping is \( C_m = \eta K / \omega \) where \( \eta \) is the loss factor due to damping. The equation of motion in the frequency domain, equation 2.22, is often summarized as

\[ K_d(\omega) u = F, \quad (2.24) \]

where \( K_d(\omega) \) is the dynamic stiffness matrix. \( F \) and \( \omega \) are the amplitude vector and the circular frequency of the applied harmonic load. (Chopra, 2007)

### 2.3.4 Radiating Boundary

During a dynamic analysis, a wave propagating against the boundary will reflect into the discretized domain. Popular methods for dealing with this effect are the plane wave approximation, the viscous damping boundary method, perfectly matched layers and infinite elements (Ross, 2004). In this thesis, the viscous damping boundary method that was developed by Lysmer and Kuhlemeyer (1969) has been implemented. They proposed that the radiating boundary is taken into account by adding an extra damping matrix to the standard discretized equation of motion, see equation 2.22. The damping matrix is built up by viscous dashpots, as in equation 2.7, at the boundary edge with a damping coefficient determined by the primary, \( V_p \), or secondary, \( V_s \), wave speed and soil material density, \( \rho \), as

\[ c_{ii} = \int_A a \rho V_p \, dA \quad (2.25) \]

in the horizontal direction and

\[ c_{ii} = \int_A b \rho V_s \, dA \quad (2.26) \]

in the vertical and horizontal directions. The concept is illustrated in figure 2.3 with the different dashpots directions shown. The dimensionless parameters \( a \) and \( b \) are chosen as one when small wave inclination angles are expected. The damping magnitude of the dashpot is integrated over the element face area, \( A \).
CHAPTER 2. THEORETICAL BACKGROUND

2.3.5 Wave Propagation in Elastic Medium

When it comes to structural dynamics, many of the practical applications of elastic wave propagation phenomenas, such as a train passage over a bridge, are sufficient to be analyzed in the elastic stress range of the material, as described in section 2.1.1.

In an infinite elastic three-dimensional space, two types of waves are expected, dilatational and distortional. The dilatational wave has a particle movement in the direction of the propagating wave. In contrary, the distortional wave has a particle movement perpendicular to the direction of the propagating wave. The dilatational wave is found to be faster then the distortional and they are therefore often called primary and secondary waves. Derived from the Navier’s equations of flow, the primary wave speed in a elastic material is found to be

\[ V_p = \sqrt{\frac{(1 - \nu)E}{(1 - 2\nu)(1 + \nu)\rho}} \]  

(2.27)

and the secondary wave speed is

\[ V_s = \sqrt{\frac{E}{2(1 + \nu)\rho}}, \]  

(2.28)

where \(E\), \(\nu\) and \(\rho\) is the Young’s modulus, Poisson’s ratio and density of the elastic material, respectively. In an elastic semi-infinite media, or half-space, other waves also exist. The most important wave is a surface wave called the Rayleigh wave. It has a lower wave speed then both the primary and secondary waves but with significantly larger amplitudes. It is also a combination of dilatational and distortional movement in the solid. (Graff, 1991)

Discretization with respect to wave velocities

A proper mesh has to be chosen so that the computational effort is limited but also so that the discretization can exhibit the proper behavior of both short and long
waves. The short waves will diminish rapidly close to the applied excitation, due to material damping, and therefore require a finer mesh close to the center of the domain. The general rule of thumb for choosing the proper mesh, is that one wave length should be represented by at least 8 elements. For the longer waves, at least a half wave should be able to be in the computational domain. The wave length, \( \lambda \), is defined as

\[ \lambda = \frac{V}{f}, \tag{2.29} \]

where \( V \) is the wave speed and \( f \) is the frequency of the wave. (Ülker-Kaustell, 2009)

2.4 Parallel FEA

There are several reasons for why numerical analysis should be performed in parallel. The original need arise from when research subject are too large and complex to be reliably predicted by theory or when they are too dangerous or expensive to be investigated in laboratory. Gropp et al. (2000) mention that the development of parallel computations arise from several aspects:

- to save computational time,
- analyze larger problems increase the demand on the available RAM and
- physical hardware limitations such as the speed of light and heat dissipation.

It is also more economical to make multi-core processors with more then one independent Central Processing Unit, CPU, than one single high speed processor. The cost increase is more rapid than the increase in processor power for a single CPU. Different models of parallel computations is today used in modern desktop computers or even smartphones. Many of the above stated reasons for parallelization fits well into the research about piles and pile groups, where the difficulties related to this subject are described in chapter 2.2. Some example of tasks that are suitable for parallelization in typical FEA, as described in section 2.3, are when

- calculating element matrices,
- solving the system of equations,
- calculating element stress and strain,

and this can be utilized especially for large-scale problems, but it can be an advantage even for rather small FEA problems. To perform this, different tools can be used. When separating the workload on more than one process, they need to be able to communicate with each other. One available standard for doing this is the Message Passing Interface, MPI, that is going to be described in section 2.4.2. An
other important part is how the solution to the problem is found. Two different types of problems are solved in this thesis and for doing this, two different tools are used. Iterative methods can be used for finding a solution to a large systems of equations and one available tool for this is the Portable, Extensible Toolkit for Scientific Computation, PETSc, and it will be described in section 2.4.3. A other type of problems is when the questing is to find eigenvalues of a discretized model, and for that purpose, the Scalable Library for Eigenvalue Problem Computations, SLEPc, has been used. It will be described in section 2.4.4.

For complex indefinite problems, common in structural vibration analysis, iterative methods may not converge. Due to this fact, also different direct methods of solving a system of equations has been adopted, namely the Multifrontal Massively Parallel Solver, MUMPS, and Distributed SuperLU with MPI, SuperLU_DIST. They will be described in sections 2.4.5 and 2.4.6. But first, a short summary of the modern object-oriented programming technique will be presented.

### 2.4.1 Object-oriented Programming

Most modern programming languages use an object-oriented approach. This technique is usually referred to as Object-oriented Programming, OOP. By using a object-oriented approach, more organized programs that are easily expanded can be written.

Objects are formed in classes and can be anything from abstract mathematical objects, such as matrices, to real world objects. The objects can be given attributes and methods callable from other classes. The matrix object mentioned earlier can be asked for its size or other internal information, depending on what functions are given to the object. The objects internal information and algorithms cannot be altered by other objects and this helps both the programmer and the object from non-intentional changes. One other important feature with objects is that they can through inheritance share methods with a base class. C++ is the object-oriented extension of ordinary C. It also exhibits the same advantages as C, for instance when it comes to efficiency and speed (Olsson, 1997).

### 2.4.2 MPI

MPI is an application programming interface for communication between processors. The MPI library can be linked from programs written and compiled with ordinary compilers in Fortran, C or C++. It consists of basic routines for sending and receiving information, to more advance routines on for example handling of new data types. It is therefore by implementing the MPI library possible to distribute workload on more then one CPU. Other models for parallel computations exist, such as shared memory or threads, but the message passing model has become widely popular.

Different models can also be combined, as usual for modern computers. The reason
for this, that has been describe in the beginning in this chapter, is due to the limitations that single machines, with one high performing or multiple CPUs, has shown when it comes to large-scale parallelization. In figure 2.4, a principal illustration of the cluster model, where shared memory machines are connected through a network, can be found. Some of the advantages that the message passing model is known for is that it is universal, expressive, the ease of debugging and also its performance. (Gropp et al., 2000)

![Network Process Memory 1]

Figure 2.4: The cluster model with message passing. From Gropp et al. (2000)

### 2.4.3 PETSc

PETSc is a set of libraries that includes different types of mathematical routines, such as vector and matrix assembly and equation solvers. It has been built with full implementation of MPI and most of the routines in PETSc are therefore scalable for parallel computations. It is callable from Fortran, C, C++ and Python (Balay et al., 2011).

Several different methods for solving equation systems are included in PETSc’s numerical libraries. Four libraries are of particular importance for this thesis: libraries for handling vectors and matrices, the Krylov Subspace Methods and preconditioners. Some of the different iterative subspace methods in PETSc (Balay et al., 2011, p. 75) include

- **Richardson**
- **Chebychev**
- **Conjugate Gradient Method, CG**
- **Minimal Residual Method, MINRES**
- **Generalized Minimal Residual Method, GMRES**
- **Biconjugate Gradient Method, BiCG**
Some of the different preconditioners in PETSc (Balay et al., 2011, p. 78) include

- Jacobi
- Successive over-relaxation, SOR
- LU factorization, LU
- Block Jacobi,
- Incomplete LU factorization, ILU
- Incomplete Cholesky factorization, ICC
- Additive Schwarz method, ASM

In section 2.5.2 and 2.5.3 the different methods and preconditioners used in this thesis will be described in more detail.

Criteria for convergence

PETSc have several different criteria to determine if a linear equation system has converged or diverged. They following reasons are given by PETSc:

- Relative tolerance - norm of residual is less than norm of right hand side by a predefined factor, $rtol$,
  \[ \| r_k \|_2 < rtol \cdot \| b \|_2. \]  (2.30)
- Absolut tolerance - norm of residual is less than a predefined constant, $atol$,
  \[ \| r_k \|_2 < atol. \]  (2.31)
- Other - small step length or negative curvature cause solution to converge.
- Maximum iterations - if the predefined value of maximum iterations is reached, solution has diverged.
- Divergence tolerance - norm of residual is more than norm of right hand side by a predefined factor, $dtol$,
  \[ \| r_k \|_2 > dtol \cdot \| b \|_2. \]  (2.32)
- Residual NaN - diverged due to NaN in residual.
- Solver breakdown - singular matrix or preconditioner. Solution has diverged.

In equation 2.30, 2.31 and 2.32 $r_k$ is the residual of the solution at iteration $k$ and $b$ is the right hand side. $\| x \|_2$ is the $l_2$ norm of complex vector $x$. The $l_2$ norm is special case of the Hölder norms with $p = 2$ in equation 2.39. (Balay et al., 2011, p. 71-75)
Choice of iterative method

One important part in using PETSc is to use the most appropriate method of solving the system of equations, choosing the "best" preconditioner and setting a sufficient relative tolerance, \textit{rtol}, for convergence. According to the PETSc manual (Balay et al., 2011, p. 163), it is recommended to perform numerical experiments to find the most appropriate set of methods for the problem at hand. Two different criteria has been used in this thesis, they are a combination of

- sufficient convergence of the solution and
- minimizing the computational time.

2.4.4 SLEPc

SLEPc is an software package for solving of large sparse eigenvalue problems on parallel computers. It has been built on top of the PETSc package and supports MPI. The main purpose of SLEPc is to solve eigenvalue problems on the standard form

\[ A x = \lambda x, \tag{2.33} \]

where \( A \in \mathbb{C}^{n \times n} \), \( x \in \mathbb{C}^{n} \) and \( \lambda \in \mathbb{C} \). Here, the scalar \( \lambda \) is the eigenvalue and vector \( x \) is the corresponding eigenvector. It can also solve problems on the generalized form

\[ A x = \lambda B x, \tag{2.34} \]

where \( B \in \mathbb{C}^{n \times n} \) (Campos et al., 2011, p. 17-18).

2.4.5 MUMPS

MUMPS is a software package for finding direct solutions to systems of linear equations on parallel computers. It can be used together with PETSc and supports MPI. The main purpose of MUMPS is to solve

\[ A x = b, \tag{2.35} \]

where \( A \in \mathbb{C}^{n \times n} \) and \( x, b \in \mathbb{C}^{n} \), by using a multifrontal approach of direct factorization. The multifrontal approach arise when the original matrix has been distributed among processors following a mapping called the dependency graph of factorization, or the elimination tree. The elimination tree can show independency between distributed tasks among processors and therefore allow for multiple factorization to be performed simultaneously. Depending on the pattern of the matrix, the factorization is performed as

\[ A = LU \quad \text{or} \quad A = LDL^T \tag{2.36} \]

for unsymmetric and symmetric matrices, respectively, where \( L \) and \( U \) is the lower and upper triangular matrix and \( D \) is the block diagonal. The direct solution is found through three main steps:
CHAPTER 2. THEORETICAL BACKGROUND

- **Analysis** - ordering and symbolic factorization
- **Factorization** - direct factorization is performed on the preprocessed matrix
- **Solution** - the solutions is found through forward and subsequently backward elimination

Each of these steps may be called independently. In this thesis, MUMPS has only been used in collaboration with PETSc, so no explicit calls to MUMPS are necessary. The control parameters used can be called at runtime by certain PETSc functions (Amestoy et al., 2011, p. 4).

### 2.4.6 Distributed SuperLU

Distributed SuperLU with MPI is one of three libraries in the SuperLU direct solver package. This version of SuperLU uses distributed memory routines to find direct solutions to systems of linear equations, as equation 2.35. SuperLU uses MPI as the communicating interface and it can also be used as an external solver with PETSc. It performs a scalable

$$A = LU$$  \hfill (2.37)

factorization through sparse Gaussian elimination (Li et al., 2011). One important feature with Distributed SuperLU is that it uses special functions for splitting the matrix among the processors, namely a 2D block-cyclic format to distribute the matrices. The block size depends on the sparsity structure of triangular matrix \(L\) and \(U\) and can therefore be of varying sizes. The SuperLU algorithm for solving system 2.35 consists of two major steps of sparse Gaussian elimination:

1. **Factorization** - compute triangular factorization as

   $$P_rD_rAD_cP_c = LU$$  \hfill (2.38a)

   where \(D_r, D_c\) are diagonal matrices to equilibrate the system. \(P_r, P_c\) are permutation matrices chosen to enhance sparsity, numerical stability and parallelism.

2. **Solution** - solve system 2.35 by evaluation of

   $$x = A^{-1}b = D_c(P_c(U^{-1}(L^{-1}(P_r(D_rb))))))$$  \hfill (2.38b)

   where a multiplication from right, \((D_r, b)\), to left is performed in the last expression. Multiplying the inverse of matrices \(L\) and \(U\) means solving the two triangular systems.
2.5 Iterative Methods for Solving Linear Systems

It is important to understand the background in linear algebra when using iterative methods to solve large sparse linear systems of equations. Some, but not all, relevant background will be described in section 2.5.1.

One can divide the available solvers for linear systems of equations in two main categories. The direct methods and the iterative methods. There is one major distinction between the two categories of solvers. It is that a direct method modifies the coefficient matrix, $A$, but the iterative methods does not. A direct method use reordering schemes for finding the exact solution, but it should be mentioned that this is not valid under floating point arithmetics. If it is possible to use a direct solution method, it is recommended. But the direct method can, because of fill-in during the factorization, make the matrix go from being sparse to dense and this usually requires large memory allocations. The iterative methods use sparse matrix-vector multiplication to solve the system and one limitation is that it may not converge. Iterative methods will be described in section 2.5.2. One way of improving the rate of convergence for an iterative method is to use a so called preconditioner to modify the coefficient matrix. If the preconditioner is successful, it should reduce the condition number of the coefficient matrix resulting in an system that is easier to solve. Common preconditioners will be described in 2.5.3 (Greenbaum, 1997).

Section 2.5.4 will address the difficulties arising when trying to solve a non-linearly parameter-dependent system of equations.

2.5.1 Background in Relevant Linear Algebra

In this section some relevant linear algebra will be presented. The reader is referred to basic literature in linear algebra for a more comprehensive understanding of these different concepts. This section has been written using Greenbaum (1997) and Saad (2003) as references if nothing else is stated.

Vector norm and inner product

The inner product of two vectors, $a, b \in \mathbb{R}^n$, is the sum of each element of those vectors multiplied with each other. The inner product produces a scalar. A norm is a function that assigns a strictly positive length or size to all vectors in a vector space, other than the zero vector. The vector $p$-norm is calculated as

$$\|x\|_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$  \hspace{1cm} (2.39)

and can be understood as just an expansion of Pythagoras theorem in multiple dimensions.
Properties of sparse matrices

One common notation when dealing with iterative methods is that the matrix should be considered sparse. But there is no sharp limit between what is a sparse matrix compared to a dense matrix. One limit could be where it is possible to take advantage of the actual quantity of non-zero elements in the sparse matrices.

\[
\begin{pmatrix}
\begin{array}{cccc}
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\end{array}
\end{pmatrix}
\begin{pmatrix}
\begin{array}{cccc}
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\ast & \ast & \ast & \ast \\
\end{array}
\end{pmatrix}
\]

(a) \hspace{1cm} (b)

Figure 2.5: Difference between a (a) dense and (b) sparse matrix of the same size. Every item illustrates a non-zero element.

One advantage with large sparse matrices is that it is possible to only store the non-zero elements of the matrix. This will save storage space in the RAM compared to saving all entries in the matrix. Different formats are available for storing matrices. A common way is the Compressed Sparse Row, CSR, format. The matrix is then stored in three arrays, each representing row offsets, column indices and values. The row offset array contains information on what row element \(n\) is on and the column indices starts on. In this way, it is possible to reduce the size of the row offset array, to the same length as the amount of rows in the matrix, compared to storing all positions of non-zero elements. If the position of all non-zero elements are stored, the format is simply called a Coordinate list. An illustration of the small sparse matrix, where all non-zero elements have one as their value, in figure 2.5(b) in the CSR format would be:

\[
\begin{pmatrix}
1 & 3 & 5 & 7 & 9 \\
1 & 4 & 2 & 4 & 3 \\
1 & 1 & 1 & 1 & 1 \\
\end{pmatrix},
\]

where the first row is the row offset, second row is column indices and last row is the values. CSR is fully implemented in PETSc for both sequential and parallel matrices (Balay et al., 2011).

Definite matrices

Matrices can be classified in terms of the quadratic from \(u^T A u\) and the matrix \(A\) is said to be:

- Positive-definite iff \(u^T A u > 0\) and \(u \neq 0\),
- Negative-definite iff \(u^T A u < 0\) and \(u \neq 0\),
2.5. ITERATIVE METHODS FOR SOLVING LINEAR SYSTEMS

- **Positive-semidefinite** iff $u^T A u \geq 0$,
- **Negative-semidefinite** iff $u^T A u \leq 0$,

and $\forall u \in \mathbb{R}^n$. For a complex matrix the above definitions would imply that $A$ is Hermitian and $\forall u \in \mathbb{C}^n$. A matrix is said to be Hermitian if and only if

$$A^H = A,$$

where $^H$ denotes the Hermitian transpose, that is $A^H_{i,j} = \bar{A}_{j,i}$, where the bar denotes the complex conjugate. If none of the above stated definitions apply, one can clearly see that if the matrix has both positive and negative eigenvalues, and the matrix is then classified as **indefinite**. This is a property that is very important when choosing the method that is used for solving a system of equation related to the matrix.

**Linear span**

A linear span is a vector space composed of all linear combinations of a set of vectors. It is defined for the set $V$ as

$$\text{span} \{V\} = \text{span} \{v_1, ..., v_n\} = \{c_1 v_1 + ... + c_n v_n \mid c_1, ..., c_n \in \mathbb{R}\},$$

where $n$ is the size of the span.

**Linear subspace**

If a subset of $\mathbb{R}^n$ contains the zero vector, is closed under addition and closed under multiplication then it is also a subspace of $\mathbb{R}^n$. The subset is closed under addition if two vectors in the set produces a new vector in the subset. Closed under multiplication states that every scalar multiplied with a vector in the subset will produce a new vector in the same subset. The **basis** for a subspace is the minimum set of vectors that are needed to span that subspace. An example of a commonly used basis is the **standard basis** in Cartesian coordinates: the unit vectors $\hat{i}$, $\hat{j}$ and $\hat{k}$ as

$$\hat{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \hat{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \hat{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$ 

**Orthogonality**

Two vectors in $\mathbb{R}^2$ are said to be **orthogonal** if they are perpendicular. This means that their dot product is equal to zero and this definition can be extended to $\mathbb{R}^n$. In a subspace of $\mathbb{R}^n$ the **orthogonal complement** of a vector form a plane in that space. That is for a subspace $V$ of $\mathbb{R}^n$,

$$V^\perp = \{x \in \mathbb{R}^n \mid x \cdot v = 0 \ \forall v \in V\}.$$
CHAPTER 2. THEORETICAL BACKGROUND

Projection

A projection is a transformation of vectors from one subset onto another subset with parallel lines. In a homogeneous subspace of $\mathbb{R}^2$, $V$, the projection of a vector $x$ onto the subspace would mean that the part of the vector parallel to the subspace would form the projection, $\text{proj}_V(x)$.

Spectrum

All eigenvalues, $\lambda$, corresponding to a matrix, $A$, is the spectrum of that matrix. The eigenvalues should satisfy

$$Ax = \lambda x,$$

where $x$ is the associated eigenvector to the eigenvalue $\lambda$. If $A \in \mathbb{R}^{2 \times 2}$ and $x \in \mathbb{R}^2$, then one can clearly see that $\lambda \in \mathbb{R}$ is only a scaling that doesn’t change direction, just the magnitude, of the vector $x$ when the linear transformation $Ax$ is performed. This is the definition of an eigenvalue. Amongst other things, eigenvectors form good basis for transformations in $\mathbb{R}^n$ with $A \in \mathbb{R}^{n \times n}$.

In practical engineering problems related to vibrations, mode superposition analysis use eigenvectors to change the basis of the global dynamic equilibrium equation. The problem is then solved for generalized displacements which can be very favorable because only a few modes are sufficient to evaluate the response of the system. (Chopra, 2007)

2.5.2 Krylov Subspace Methods

The Krylov subspace methods is a general name for all iterative methods using a Krylov subspace to approximate the solution of a linear system of equations on the form

$$Ax = b$$

with the Krylov subspace defined as

$$K_N(A, b) = \text{span}\{b, Ab, A^2b, A^3b, ..., A^{N-1}b\}, \quad (2.40)$$

where $N$ is the size of the subspace. As described in section 2.5.1, a span is just all linear combinations of the different vectors in the span. So, if the size of the Krylov subspace is chosen as the number of unknowns in the linear system of equations, the subspace is all the linear combinations available in the span. Because the subspace form a basis for the system of equations the solution will always be in the subspace. This is the reason why a Krylov subspace is used in the iterative method. It limits the search to only possible linear combinations between the coefficient matrix and the right hand side. (Greenbaum, 1997)

The difference between the available iterative methods that uses the Krylov subspace is how they define what the "best" solution is in the subspace. Two of the most commonly used methods will be described in the following sections.
In should also be noticed that the iterative method may have reached sufficient accuracy earlier than what was expected. It is therefore important to analyze the spectrum of the operator, where the operator is the transformation matrix, to understand where the solution can be found. This is a difficult task and from a practical standpoint, a convergence analysis is generally perform instead.

**Conjugate Gradient Method**

The *Conjugate Gradient Method*, CG, is one of the best methods of finding an iterative solution when the matrix is sparse, symmetric and also positive-definite. This is made obvious when the algorithm is examined. At each iteration the $A$-norm of the error, $\|x - x_k\|_A$, is minimized in the Krylov subspace and the initial guess is then updated as

$$x_k = x_0 + K_k(A, r_0),$$  \hspace{1cm} (2.41)

where $r_0 = b - Ax_0$ is the initial residual and $k$ is the current iteration. This method usually converges faster then other iterative methods but have limitations due to the necessary matrix properties. (Greenbaum, 1997)

**Generalized Minimal Residual Method**

The *Generalized Minimal Residual Method*, GMRES, builds the Krylov subspace by adding one vector at each iteration by the Arnoldi method. The Arnoldi method can be used to approximate eigenvalues and uses a modified Gram-Schmidt process that constructs an orthogonal basis for the Krylov subspace. It then determines the approximate solution by minimizing the norm of the associated residual, $\|b - Ax_k\|$. Each iteration can be summarized as just

$$x_k \in x_0 + \mathcal{K}_k(A, r_0),$$  \hspace{1cm} (2.42)

where $r_0 = b - Ax_0$ is the initial residual and $k$ is the current iteration. Usually, to reduce memory consumption, the process is restarted at a predetermined size of the Krylov subspace, $N$. A new residual is then computed and used in the subsequent iterations. (Saad, 2003)

### 2.5.3 Preconditioners

When using preconditioners together with an iterative method for solving a linear systems of equations the goal is to reduce the condition number of the matrix. This will help steer the iterative method in the direction of the solution and therefore cause the system to converge faster. The precondition matrix, $M$, can be applied on the left, right or both sides of the original equation. For left preconditioning, the preconditioned linear system would look like

$$M^{-1}(Ax - b) = 0.$$  \hspace{1cm} (2.43)
Two different theoretical criteras are used when the preconditioner is chosen: the linear system with the preconditioning matrix should be easier to solve and if it is multiplied with the coefficient matrix, the result should approximate the identity. Several different methods are available for choosing the preconditioning matrix and to illustrate how it can be chosen, two different methods will be described in the following sections. (Greenbaum, 1997)

**Jacobi**

The simplest method available is is to take the inverse of the diagonal of the coefficient matrix. That is

\[
M = \begin{cases} 
A_{i,j} & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}
\]

This method can also be used when solving problems in parallel. When using block Jacobi preconditioner, the system matrix is divided in blocks. Each block is then precondition separately. This is a simple way of preconditioning in parallel, because each block is calculated on one computational node. (Greenbaum, 1997)

**Incomplete LU factorization**

LU factorization is a widely used direct method for solving a system of equations. The coefficient matrix is factorized into a lower triangular matrix, \( L \), and a upper triangular matrix \( U \) as

\[
A = LU \tag{2.44a}
\]

and then the system is solved through first forward substitution and then back substitution, as

\[
Lz = b, \tag{2.44b}
\]

\[
Ux = z, \tag{2.44c}
\]

where \( z \) is a vector used to store the results from the forward substitution and then used as the right-hand side for the back substitution. The important thing to notice with the forward and back substitution is that all the values in \( z \) and \( x \) is known, it is just a matter of calculating them. LU factorization can be very powerful when the system is solved for many different right-hand sides, because the lower and upper triangular matrix does not change.

From the LU factorization, a popular preconditioner has been derived, the *Incomplete LU Factorization*, ILU. One important feature with ILU is that it retains the non-zero structure of the coefficient matrix, so a sparse matrix will remain sparse. This is obtained by dropping the elements that become non-zero during the factorization if they deviate from the chosen non-zero pattern. Here, the decomposition results in

\[
A = LU - R, \tag{2.44d}
\]
where \( L, U \) is the sparse lower and upper triangular matrix of the coefficient matrix. Matrix \( R \) contains all the dropped elements during the incomplete factorization. The zero-pattern used need to be chosen in advance. If the set of the non-zero pattern, \( P \), is chosen to be the same as in \( A \) then the preconditioning method is called zero fill-in ILU, ILU(0), and

\[
P \subset \{(i,j) \mid i \neq j; \ 1 \leq i,j \leq n\}.
\]

If the level of fill allowed during the factorization is denoted \( p \), then the preconditioner is denoted ILU(\( p \)). (Saad, 2003)

### 2.5.4 Parameter-Dependent System of Equations

This section will address the numerical solution of non-linearly parameter-dependent system of equations. The following system is considered with a wide range of values for the dependent parameter \( \lambda \),

\[
L(\lambda)x = b, \quad L(\lambda) = \lambda^2 A + \lambda B + C,
\]

where \( x, b \in \mathbb{C}^n \) and \( A, B, C \in \mathbb{C}^{n \times n} \). One can clearly see that this is the valid case for the matrix system 2.22 if

\[
\lambda = \omega, \quad A = -M, \quad B = iC, \quad C = K.
\]

(2.46)

It has been pointed out by Ernst and Gander (2011) that standard iterative methods for solving linear systems can be very hard when the coefficient matrix is indefinite. They show that for a standard method such as GMRES, see section 2.5.2, the convergence stagnates. Even the standard preconditioners cannot reduce the condition number of the system, it can actually be worse to use a standard preconditioner. They also show that the system is harder to solve for higher frequencies, \( \omega \). Because of this, they describe that different, specialized and more advanced, iterative methods and preconditioners could solve this type of problem but they are very sophisticated and can be hard to implement.

However, earlier studies show that numerical solution of non-linearly parameter dependent systems of equations, as system 2.45, can be solved with iterative methods on a modified matrix formulation. Simoncini and Perotti (2002) suggests that a linearization of the problem should be performed, then, a system twice the size of the original needs to be solved. The system would after linearization look like

\[
\left( \begin{bmatrix} B & C \\ C^T & 0 \end{bmatrix} + \lambda \begin{bmatrix} A & 0 \\ 0 & -C^T \end{bmatrix} \right) \begin{bmatrix} \lambda x \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}.
\]

(2.47)

Feriani et al. (2000) showed that taking the inverse formulation of system 2.47 could outperform the direct approach shown here. The inverse formulation would in the structural dynamic case solve for acceleration rather then the displacements, by letting \( a = -\omega^2 u \).

27
A, so called, inner-outer procedure where a range of values of $\lambda$ is solved by only one vector projection is suggest by Gu and Simoncini (2005). They propose that the system 2.47 is shifted and because of this, the Krylov subspace that is built up by the method is not parameter-dependent. Their work also address the possibility of a parameter-dependent right-hand-side.

For further reading, a great summary by Simoncini and Szyld (2007) is recommended. It includes many references and also background on iterative methods in general. It should be pointed out that iterative methods are very favorable for this type of problem if an efficient method can be used. This is because if a direct method is used the factorization of the coefficient matrix needs to be performed for every value of $\lambda$. If large-scale problems in combination with a very large range of dependent values should be solved, this procedure is very costly. As one can understand, this is a complex subject where further research is necessary.
Chapter 3

Scripts

This chapter describes the scripts that have been written as the scope of the thesis. First, the general philosophy and some comments about the development of OOP for FEA from the literature are presented. Subsequently, the objects and PETSc-functions used are described and also the general outline of the algorithms for the scripts. The full scripts can be found in appendix B.

3.1 General Philosophy

The basics of an object-oriented finite element implementation has been under development during the last 20 years in the academic society. Early suggestion by Zimmermann et al. (1992) describe the general outline of how OOP could be implemented for FEM. Some basic algorithms are also later considered in Olsson (1997). Other examples can also be found in the literature. The approach chosen in this thesis is very problem-oriented but still follows the same philosophy. What is special with the approach used here is that all scripts can be used in parallel so that the work-load can be separated on multiple nodes in a supercomputer or a cluster of workstations, as described in section 2.4. To give the reader an understanding about how the scripts and the different classes building up them work, the following sections are dedicated to that. But first, the general philosophy will be described.

The different tasks are separated into three different categories. First, preparing the matrices and vectors in the preprocess from an externally created mesh, then finding the solution and last, visualizing the results. The mesh has for obvious verification reasons been generated in ABAQUS. The different parts has been visualized with input and output in figure 3.1. Each process is communicating with datastructures and functions in PETSc. With this approach, the scripts can easily be implemented to one executable file and also extended for other purposes. For instance, the author has written one script for static analysis and one for eigenvalue problems that uses the same pre- and postprocessor-script, but with a different solver-script. The eigenvalue solver script has been visualized in figure 3.2.
CHAPTER 3. SCRIPTS

Preprocess
PETSc
Material properties
Boundary conditions
Loads
Mesh
Matrices: $K, M$
Vectors: $F, C$

Steady-state (SS-FEA)
PETSc
Matrices: $K, M$
Vectors: $F, C$
Vector: $u$

Postprocess
PETSc
Mesh
Boundary conditions
Vectors: $F, C, u$
Visualization

Figure 3.1: Illustration of the different scripts developed with input and output

Static
PETSc
Matrices: $K$
Vectors: $F$
Vector: $u$

Eigenvalue (EPS)
SLEPc
Matrices: $K, M$
Scalar: $\lambda$
Vector: $x$

Figure 3.2: Illustration of the static and EPS-script with input and output
Most of the functionality of the executable scripts can be altered at run-time. For the preprocess, what system matrices and vectors that should be computed may be specified and also what directory they should be stored in. For the steady-state script, obvious things such as what range of variables and what matrices should be used can be decided at run-time.

3.1.1 Input File Properties

The input file is created in an external software for mesh creation. Currently, the input file properties and preprocessor are adapted to use with the commercial software package *ABAQUS* input file generator. Other, open programs could easily be implemented if desired but it is recommended that the same type of sets should be used. The type of input file could then easily be determined by a run-time option. Besides the obvious things such as nodes with Cartesian coordinates and elements with associated nodes, the input file should also consist of a number of sets that define the actual problem. Four different categories of sets are used:

* set, nset = LockedXYZ
* Elset, elset = Concrete, generate
* Surface, type = ELEMENT, name = DomainEdge
* Surface, type = ELEMENT, name = Load

This approach makes it possible to specify the sets when creating the mesh. Here, only four different sets are used but it can easily be expanded to a more general approach. Node set 3.1 is used to define what degrees of freedom that should be locked in the x-, y- and z-direction. Element set 3.2 is used to define what elements should have the material properties of concrete, otherwise the standard properties are that of soil. The two surface sets (3.3, 3.4) are used to define the surfaces at the edge of the domain, for the radiating boundary, and on what surfaces the loads should be applied.

3.2 Objects

Except from the main file, the mesh object is the most important, holding all the information about the model. The whole structure of the executable pre- and post-process scripts can be found in figure 3.3. The main file only needs to hold information about the mesh, load and material objects. In the following sections, the different objects will be described.
CHAPTER 3. SCRIPTS

Mesh

The mesh object holds information on what nodes, elements and faces that are in the mesh. It also holds the appropriate mapping of nodes to degrees-of-freedom and the degrees that are active. When the mesh is created, all associated objects are created and stored in appropriate containers.

Element

The element object is created with a numbering and containers for its nodes, faces and a pointer to the associated material. Different types of elements are inherited from this base class. This has a great advantage because independently of how the element stiffness and mass matrices are calculated, they can be returned with the same function. Only linear elements have been fully implemented at this stage; linear tetrahedron and hexahedron elements. The tetrahedron element can be used for dynamic analysis.

Node

The node object is quite simple. It knows its number and also its Cartesian coordinates and can return them if asked.

Figure 3.3: Illustration of how the objects are related to each other for the executable preprocess script. The same structure applies for the postprocess script. Dashed lines illustrate inheritance.
3.3 PETSc Functions

Face

In the same way as the node object, the face objects is also very simple. When it is created, it is given a number and a container for the associated nodes.

Load

Three different objects are inherited from the load class: body force, surface traction and boundary damping. These loads are applied during the preprocess when the right-hand-side or damping vector is assembled. The loads are applied on elements-sets that are specified in the input file and uses pointers to the element to get the appropriate distribution of the load.

Material

Holds information about material properties. Different materials are inherited from the material class, such as steel, soil or concrete.

3.3 PETSc Functions

Important procedures with PETSc functions used in the different scripts will be pointed out in this section. Repeated arguments have been excluded for some functions. Version 3.2 p5 of PETSc was used. If any other version is used, these functions and algorithms may have to be updated.

3.3.1 Matrix and Vector Assembly

The system matrices and vectors are assembled in the preprocessing-script. They are created with the following functions:

\[
\begin{align*}
\text{MatCreateMPIAIJ}(\text{MPI_Comm}, \text{PetscInt}, \ldots, \text{Mat}^*) & \quad (3.5) \\
\text{MatSetValues}(	ext{Mat}, \text{PetscInt}, \ldots, \text{PetscScalar}, \text{InsertMode}) & \quad (3.6) \\
\text{MatAssemblyBegin}(	ext{Mat}, \text{MatAssemblyType}) & \quad (3.7) \\
\text{MatAssemblyEnd}(	ext{Mat}, \text{MatAssemblyType}) & \quad (3.8) \\
\text{MatOut}(	ext{Mat}, \text{string}) & \quad (3.9) \\
\text{MatDestroy}(	ext{Mat}^*) & \quad (3.10)
\end{align*}
\]

Function 3.6 is looped over all elements. Function 3.9 is a user-defined function that prints out the matrix in binary format in the directory specified with the string-argument. The vector assembling follows almost the same procedure. The
main difference is that two different user-defined functions are used to set values, depending on the purpose. They are the following for loading or radiating damping:

VecSetMapLoad(Vec*, PetscInt, Mesh*, multimap, ..., double, ...) (3.11)
VecSetMapDamp(Vec*, PetscInt, Mesh*, multimap, ..., double, ...) (3.12)

The vector object has similar functions for creation (3.5), assembly (3.7,3.8), output (3.9) and destruction (3.10) as the matrix object. It should be pointed out that the assembly routine (3.7,3.8) is the one distributing the matrices and vectors among nodes for distributed data.

### 3.3.2 Steady-state Solution Procedure

In the steady-state-script the procedure is separated in two parts: first, all the necessary data is loaded into the RAM and then a loop over the range of frequencies of interest is performed where the problem is solved at each frequency. The following functions are used for the first part:

MatCreateMPIAIJ(MPI_Comm, PetscInt, ..., Mat*) (3.13)
MatIn(Mat, string) (3.14)
VecCreate(MPI_Comm, PetscInt, ..., Vec*) (3.15)
VecIn(Vec, string) (3.16)
KSPCreate(MPI_Comm, KSP*) (3.17)
MatScale(Mat, PetscScalar) (3.18)

Functions 3.13-3.16 loads the matrices and vectors. Function 3.17 creates the solution object and function 3.18 then scales the stiffness matrix so that the structural damping is present. For the second part, the following functions are used:

MatDuplicate(Mat, MatDuplicateOption, Mat*) (3.19)
MatAXPY(Mat, PetscScalar, Mat, MatStructure) (3.20)
VecDuplicate(Vec, Vec*) (3.21)
VecCopy(Vec, Vec) (3.22)
VecScale(Vec, PetscScalar) (3.23)
MatDiagonalSet(Mat, Vec, InsertMode) (3.24)
KSPSetOperators(KSP, Mat, Mat, MatStructure) (3.25)
KSPSolve(KSP, Vec, Vec) (3.26)
KSPOut(KSP, Mat, Vec, ..., PetscLogDouble, ..., int, char*) (3.27)
VecOut(Vec, string) (3.28)
VecDestroy(Vec*) (3.29)
MatDestroy(Mat*) (3.30)

First, the global stiffness matrix with structural damping is duplicated (3.19) and then the mass matrix is added with a scalar multiplier representing the frequency
(3.20). In the same manner, the radiating damping vector is duplicated, copied, scaled and then inserted into the dynamic stiffness matrix (3.21-3.24). The dynamic stiffness matrix is assigned to the solution object through function 3.25 and then solved with function 3.26. Information about the solution process and the solution vector is written with the two user-defined functions (3.27, 3.28).
Chapter 4

Verification

It is always important to verify numerical calculations. In this thesis, verification of the scripts developed has been done by comparison with solutions of analytical problems, semi-analytical problems or solutions obtained from commercial software. First, two different analytical solutions from the theory of elasticity is considered. Also a comparison of results obtained from an eigenvalue problem analysis has been done to those obtained from a analysis performed in a commercial software package. To verify the steady-state analysis procedure used in this thesis, a simple problem is solved and then the results is compared to those obtained from a commercial software package. Also the results from a analysis with radiating boundaries are compared against a commercial software package.

4.1 Elementary Problems of Elasticity in Three Dimensions

In the field of elastic theory, a certain number of analytical solutions to elementary problems has been derived in the literature. A few of them will be used as validation of the FEA-script developed in the scope of this thesis.

4.1.1 Stretching of a Prismatical Bar by its Own Weight

Timoshenko and Goodier (1970) has solved the differential equations of equilibrium for a prismatical bar that is elongated by its own weight. The following expressions of displacements are presented,

\[ u = -\frac{\nu \rho g}{E} xz, \quad v = -\frac{\nu \rho g}{E} yz, \quad w = \frac{\rho g}{2E} \left[ (z^2 - l^2) + \nu(x^2 + y^2) \right], \]  

(4.1)

where \( E, \nu, \rho \) and \( g \) is Young’s modulus, Poisson’s ratio, density and the ground acceleration, respectively. The prismatic bar that has been used can be found illustrated in figure 4.1 and has a width and height \( w = h = 1 \text{m} \) and length \( L = 5 \text{m} \). The material is steel with the following properties: \( E = 210 \text{GPa}, \nu = 0.30, \rho = 7800 \text{kg} \cdot \text{m}^{-3} \).
Gravity is set to $g = 10 \text{m} \cdot \text{s}^{-1}$. The results are summarized and compared to the analytical solution in Table 4.1. A small difference is found and the error is related to the error in the discretization.

![Figure 4.1: Geometry of the prismatic bar elongating by its own weight.](image)

**Table 4.1: Comparison between analytical and FEA results for a prismatic bar elongated by its own weight**

<table>
<thead>
<tr>
<th>Point $(x,y,z)$</th>
<th>$dL$</th>
<th>0.200</th>
<th>0.150</th>
<th>0.100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0,0</td>
<td>FEA</td>
<td>4.651</td>
<td>4.652</td>
<td>4.652</td>
</tr>
<tr>
<td></td>
<td>Analytical</td>
<td>4.643</td>
<td>$10^{-6}$ m</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relative error</td>
<td>0.17</td>
<td>0.19</td>
<td>0.19 %</td>
</tr>
<tr>
<td>0.5,0.5,0</td>
<td>FEA</td>
<td>4.624</td>
<td>4.626</td>
<td>4.624</td>
</tr>
<tr>
<td></td>
<td>Analytical</td>
<td>4.615</td>
<td>$10^{-6}$ m</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relative error</td>
<td>0.19</td>
<td>0.24</td>
<td>0.19 %</td>
</tr>
</tbody>
</table>

**4.1.2 Pure Bending of Prismatical Bars**

The following theory of bending has been taken from Timoshenko and Goodier (1970). They show how a prismatic bar of a elastic material will respond to pure bending. The radius of curvature of the bar undergoing pure bending moment is, well-know from elementary theory of bending, derived as

$$
\frac{1}{R} = \frac{M}{EI_y}
$$

and together with the expressions that determines the displacements of a prismatical bar

$$
\begin{align*}
    u &= -\frac{z^2 + \nu(x^2 - y^2)}{2R}, \quad v = -\frac{\nu xy}{R}, \quad w = \frac{xy}{R},
\end{align*}
$$

(4.3)
the movement of a bar can be found. The prismatic bar that has been used can be found illustrated in figure 4.2 and it has a width $w = 0.8\text{m}$, height $h = 0.4\text{m}$ and length $L = 5\text{m}$. The material is concrete with the following properties: $E = 20\text{GPa}$, $\nu = 0.25$, $\rho = 2500\text{kg} \cdot \text{m}^{-3}$. The results are summarized and compared to the analytical solution in table 4.2. A good compliance with the FEA results are found, where an increased discretization leads to convergence towards the analytical result.

![Figure 4.2: Geometry of the prismatic bar undergoing pure bending moment.](image)

<table>
<thead>
<tr>
<th>Point $(x,y,z)$</th>
<th>dL</th>
<th>0.250</th>
<th>0.200</th>
<th>0.150</th>
<th>0.100</th>
<th>0.050</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.4,0.2,5.0$</td>
<td>$y$</td>
<td>FEA</td>
<td>-32.87</td>
<td>-35.25</td>
<td>-36.90</td>
<td>-41.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Analytical</td>
<td>-46.82</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relative error</td>
<td>-42.42</td>
<td>-32.80</td>
<td>-26.87</td>
<td>-13.08</td>
<td>-3.74</td>
</tr>
<tr>
<td>$z$</td>
<td>FEA</td>
<td>2.60</td>
<td>2.80</td>
<td>2.96</td>
<td>3.30</td>
<td>3.59</td>
</tr>
<tr>
<td></td>
<td>Analytical</td>
<td>3.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relative error</td>
<td>-44.12</td>
<td>-33.93</td>
<td>-26.69</td>
<td>-13.60</td>
<td>-4.43</td>
</tr>
<tr>
<td>$0.4,0.2,2.5$</td>
<td>$y$</td>
<td>FEA</td>
<td>-8.25</td>
<td>-8.77</td>
<td>-9.13</td>
<td>-10.28</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Analytical</td>
<td>-11.62</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relative error</td>
<td>-40.96</td>
<td>-32.57</td>
<td>-27.24</td>
<td>-13.10</td>
<td>-3.78</td>
</tr>
<tr>
<td>$z$</td>
<td>FEA</td>
<td>1.34</td>
<td>1.40</td>
<td>1.48</td>
<td>1.66</td>
<td>1.81</td>
</tr>
<tr>
<td></td>
<td>Analytical</td>
<td>1.88</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relative error</td>
<td>-40.34</td>
<td>-33.55</td>
<td>-26.60</td>
<td>-12.95</td>
<td>-3.76</td>
</tr>
</tbody>
</table>

### 4.2 Natural Frequencies and Mode Shapes

The natural frequencies of a discretized structure can be found by solving the generalized eigenvalue problem

$$Ku = \omega^2 Mu,$$

where $u$ is the eigenvector, $\omega^2$ is the eigenvalue and $\omega$ is the corresponding natural circular frequency. By doing a comparison of the obtained eigenvalues and eigenvectors from the analysis to results from established solvers, in this thesis *ABAQUS*,

39
the derived mass and stiffness matrices can be validated. In structural dynamics, usually the lowest eigenvalues are of interest and therefore, the sixth lowest have been compared.

Two different structures were analyzed. One cantilever prismatic bar, the same as in the case with pure bending analysis in section 4.1.2, and also a bridge structure with distinct mode shapes. The bridge has one column and two spans stretching to a roller support on each side. The column is assumed to be fully clamped at the bottom. Each span of the bridge has a total length \( L = 35.0 \text{m} \) and the free height of the column \( H = 9.0 \text{m} \) at the middle. Cross section properties of the bridge deck is width \( w = 1.8 \text{m} \) and height \( h = 0.8 \text{m} \) and for the column width \( w = 1.8 \text{m} \) and height \( h = 0.5 \text{m} \). The material is concrete with the following properties: \( E = 20 \text{GPa}, \nu = 0.25, \rho = 2500 \text{kg} \cdot \text{m}^{-3} \). Both structures has been illustrated in figure 4.3. Results from the two analyses can be found in table 4.3 and 4.4. The first six mode shapes from the second analysis can be found in appendix A.

![Diagram of prismatic bar and bridge structure](image)

Figure 4.3: Geometry of (a) prismatic bar and (b) bridge structure used for the natural frequency and mode shape analysis.

<table>
<thead>
<tr>
<th>Mode</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eigenvalue</td>
<td>2353.96</td>
<td>8442.75</td>
<td>87248.88</td>
<td>214921.09</td>
<td>271564.81</td>
<td>633156.85</td>
</tr>
<tr>
<td>Frequency</td>
<td>7.722</td>
<td>14.624</td>
<td>47.011</td>
<td>73.784</td>
<td>82.939</td>
<td>126.641</td>
</tr>
<tr>
<td>ABAQUS</td>
<td>7.720</td>
<td>14.620</td>
<td>46.916</td>
<td>72.012</td>
<td>82.789</td>
<td>126.080</td>
</tr>
<tr>
<td>Relative error</td>
<td>0.03</td>
<td>0.03</td>
<td>0.20</td>
<td>2.40</td>
<td>0.18</td>
<td>0.44</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison of eigenvalues for a prismatic bar
4.3 Steady-state Analysis

A steady-state analysis of a prismatic bar has been performed. The load case was a vertical shear stress at the end of the cantilever with a real magnitude of 10kPa. The calculation were performed from 0.5Hz to 100Hz with a step of 0.5Hz. Structural damping, $\eta = 4\%$, was applied to the whole cantilever. The geometry and other material properties are the same as for the bar used in the natural frequency analysis, section 4.3. The cantilever has been illustrated in figure 4.4.

To find the proper relative tolerance and a suitable preconditioner, a convergence analysis was performed. Due to the fact that the matrix is indefinite, the best suited preconditioner was found to be a combination of ASM and block LU factorization. Three different frequencies was used in the convergence analysis for relative tolerance, $rtol$, and they can be found in figure 4.5, 4.6 and 4.7. Sufficient convergence was found with a relative tolerance of $10^{-5}$, for all investigated frequencies. The vertical response at the tip of the bar from the steady-state analysis can be found in figure 4.8. A very good compliance is found with the results from ABAQUS.
Figure 4.5: Convergence analysis at 10Hz for steady-state analysis solved with an iterative method.

Figure 4.6: Convergence analysis at 25Hz for steady-state analysis solved with an iterative method.
Figure 4.7: Convergence analysis at 50Hz for steady-state analysis solved with an iterative method.

Figure 4.8: The vertical response of the bar from steady-state analysis and compared against results from ABAQUS.
4.4 Radiating Boundary

To verify the radiating boundaries used within the scripts created, a comparison was performed with results from ABAQUS. The model consists of a linearly elastic, homogeneous, isotropic circular soil layer with a radius of 40 meter, it has been illustrated in figure 4.9. On the soil layer, a square area with side length of 2 meter and a real vertical load of 0.25Pa, resulting in a unit load of 1N, has been applied to simulate a flexible plate. The analysis was performed from 1.0Hz to 40Hz with increments of 1.0Hz. The material properties of the soil layer are the following: $E = 50\text{MPa}$, $\nu = 0.25$, $\rho = 2000\text{kg}\cdot\text{m}^{-3}$, $\eta = 4\%$. The vertical compliance functions, calculated as described in section 2.1.3, for the plate can be found in figure 4.10.

![Figure 4.9: Geometry of the circular soil layer with a flexible plate.](image)

![Figure 4.10: Comparison of the steady-state response of a plate resting on an elastic circular soil layer from the steady-state script and ABAQUS. Compliance functions for the vertical response at the center point of the plate.](image)
Chapter 5

Case Study

In this chapter, a case study on the dynamic properties of a vertical pile will be described. An introduction, section 5.1, will be followed by the used method of analysis, section 5.2, and results and discussions, section 5.3 and 5.4, upon the compared numerical and semi-analytical results.

5.1 Vertical Pile in Elastic Soil

To be able to continue the numerical studies on large and more complex pile groups, a tool has been developed as part of the scope of this thesis. As describe in section 2.2, this subjects is very complex and a lot of research has been put into understanding and predicting the dynamic behavior of pile groups. Nevertheless, a number of interesting questions that are applicable to practical engineering, are still not fully understood. This tool can therefore become useful in further analysis and to enhance the understanding of pile and pile group behavior. As described in section 2.4, this tool have features that makes it possible to use on clusters of work-stations or supercomputers. This open up the possibility to analyze large-scale problems, such as the wave phenomena of pile-soil-pile interaction.

To verify that the tool is performing according to the current theories, a case study for a vertical pile has been performed. The methodology used, results and discussions about the case study can be found in the following sections.

5.2 Method

5.2.1 Geometry and Material Properties

An illustration of the geometrical model can be found in figure 5.1. The pile is rectangular with side length, $d = 0.3\text{m}$. The vertical extent of the soil layer, $H$, has been limited to 3 meters. The material properties of the pile is concrete with $E =$
20GPa, $\rho = 2500\text{kg/m}^3$ and $\nu = 0.25$. The soil layer has the following homogeneous properties: $E = 50\text{MPa}$, $\rho = 2000\text{kg/m}^3$ and $\nu = 0.25$. The pile is assumed to be rigidly connected to the bedrock.

A continuous viscous damper has been applied at the boundary of the computational domain, as described in section 2.3.4, so that reflection at the boundary is minimized. One might think that the viscous damper are so effective that only a small radius of the circular layer of soil can be used for higher frequencies. This because of the material and radiating damping taking place. But numerical tests show that the amplitude of the waves need to have damped out in the soil for the model to behave satisfactory. Because of this, a constant rather large radius was used for all frequencies: $R = 40\text{m}$.

Figure 5.1: Geometry of the vertical pile and visco-elastic soil stratum on a rigid half-space. Figure (a) and (b) show the model in section and plan, respectively. The case with vertical load is illustrated as $F(t)$. 
5.2. METHOD

5.2.2 The Finite-element Model

The mesh was chosen with respect to the wave propagation phenomena that will occur, as describe in section 2.3.5. A high mesh density was used close to the pile and lower density closer to the edge of the domain. A linear increase in mesh size was used in between. The discretized model is shown in figure 5.2. The whole model, including the pile, is meshed with linear tetrahedron elements (C3D4).

Figure 5.2: The finite-element model used for calculating the frequency functions for the vertical pile. Figure (a) and (b) show the model from top and a section in perspective, respectively.
5.2.3 Load Cases

Four different load cases were considered: vertical, horizontal, torsional and rotational loading. The cases are illustrated in figure 5.3.

![Figure 5.3: Four different load cases; (a) vertical, (b) horizontal, (c) torsional and (d) rotational case.](image)

5.2.4 Solution Procedure

To ensure a proper result, a preliminary analysis where certain factors influencing the results was taken under consideration was performed. First, a proper mesh has to be chosen. Both static and dynamic analysis was performed to ensure convergence. It was found during the preliminary analysis that the radius of the visco-elastic stratum has a major influence on how well the viscous boundary is performing, as described in section 5.2.1. Because of this, a study on what radius is necessary was also undertaken in the preliminary analysis.

The last step of the preliminary analysis would have been a convergence analysis on relative tolerance, \(rtol\), for the iterative method. But, as were described earlier, using an iterative method for this problem are very difficult and the idea had to be abandon. Instead, a scalable direct solution method was used and the performance different was compared, MUMPS and Distributed SuperLU. It was found that the Distributed SuperLU package had the best performance and was therefore used in the subsequent calculations. As described in section 2.2.1, it has already been concluded that the dynamic behavior of single piles is constant over \(a_0 = 0.5\). Because of this, the upper frequency limit was, in this case study, set to 50 Hz \((a_0 = 0.47)\). The main three steps of the analysis were:

- Preliminary analysis of
  1. convergence of mesh,
  2. sufficient radius of model,
  3. solution method.
- Perform analysis from 1Hz to 50Hz for four different load cases.
- Extraction of results and comparison against the semi-analytical solution.
5.2.5 Calculation of the Dynamic Stiffness Function

The dynamic stiffness and equivalent viscous damping was calculated for each exciting frequency. The inverted dynamic stiffness was first calculated using

\[
\begin{align*}
    f_{zz,1} &= \frac{\text{Re}(u)}{F_{zz}}, & f_{zz,2} &= \frac{\text{Im}(u)}{F_{zz}}, \\
    f_{xx,1} &= \frac{\text{Re}(u)}{F_{xx}}, & f_{xx,2} &= \frac{\text{Im}(u)}{F_{xx}}, \\
    f_{x\psi,1} &= \frac{\text{Re}(\psi)}{F_{x\psi}}, & f_{x\psi,2} &= \frac{\text{Im}(\psi)}{F_{x\psi}}, \\
    f_{\psi\psi,1} &= \frac{\text{Re}(\psi)}{F_{\psi\psi}}, & f_{\psi\psi,2} &= \frac{\text{Im}(\psi)}{F_{\psi\psi}}
\end{align*}
\] (5.1)

where \( F_{ij} \) is the total load effect acting on the pile for the different load cases, \( ij = (x, z, \psi) \). \( u \) and \( \psi \) is the complex displacement and rotation, that varies with frequency, of the pile cap. The results from equation 5.1 is then converted to stiffness and equivalent viscous damping through equation 2.6 and then back calculated from equation 2.9 to a dimensionless form. The comparison is made with dimensionless frequency functions, calculated using equation 2.11 and 2.13.

5.3 Results

In the following section, the results form the steady-state analysis for four different load cases are presented.

5.3.1 Dynamic Stiffness Functions for Vertical Pile

Vertical, horizontal, torsional and rotational frequency functions can be found in figure 5.4, 5.5, 5.6 and 5.7, respectively. They can be extended to stiffness and damping with equation 2.9. The damping has in all cases been over-estimated, except for the torsional load case for \( a_0 > 0.15 \), compared to the approximate analytical solution. The vertical load case show one distinct soil layer resonance at \( a_0 \approx 0.14 \) but none for the torsional load case. Both the horizontal and rotational load case show several layer resonances.
Figure 5.4: Vertical stiffness as dimensionless frequency function $F_{18}$.

Figure 5.5: Horizontal stiffness as dimensionless frequency function $F_{6}$. 
5.3. RESULTS

Figure 5.6: Torsional stiffness as dimensionless frequency function $F_4$.

Figure 5.7: Rotational stiffness as dimensionless frequency function $F_2$. 
5.3.2 Direction of Soil Wave Motion

The direction of the elastic soil wave motion is different depending on the type of excitation. The different load cases with wave motion are illustrated in figure 5.8 and 5.9. It was found that from a vertical load case, only vertical motion is found in the soil along the principal axes. In the same manner, from the torsional load case only horizontal waves occur. In both the horizontal and rotational load case, vertical and horizontal waves form but in different directions.

Figure 5.8: Principal distribution of wave motion in the circular soil stratum. Upper right quarter is illustrated with wave motion along the principal axes. Figure (a) and (b) show the vertical and torsional load case, respectively.
5.4 Discussion

The approximate formulas used have been calibrated against field experiments and rigorous solutions based on mode shape analysis. It has been shown that the dynamic properties do not perfectly match between the approximate solution and the numerical steady-state analysis, SS-FEA. This corresponds to what was expected, because the approximate solution is adjusted against field experiments and mode superpositioning calculations. Larger differences have been found, in all cases, for the viscous damping than for the stiffness. This is probably because of the material damping, described in section 2.1.4, that was implemented in the scripts and used during the numerical analysis. It has later been identified that the approximate solution does not have this type of damping implemented and this can therefore explain some of the disparities in the dynamic response of the vertical pile.

One other possibility is that the material damping, assumed to 4% in the analysis, is not corresponding to the actual material damping that the approximated solution...
was verified against. A parameter variation could be performed to see the influence on material damping on the numerical results.

The quality of the mesh was found to have great influence on the numerical results. It was also found that the radius of the visco-elastic stratum is very important for the efficiency of the viscous damping boundary at the edge of the computational domain. The same radius was used in all calculations but because of the different wave motions that occur, one might have to use a larger radius for those with larger wave amplitudes and multiple wave directions. Load cases that cause wave motion in multiple directions, such as horizontal and rotational, appear to have more than one soil layer resonance. This is probably because the elastic wave frequency in the soil is different for horizontal and vertical waves.
Chapter 6

Conclusions and Suggestions for Future Research

The development of a completely new tool for dynamics analysis is a major task. This thesis form the basis of such a tool, but further development is necessary. The major outcome of this thesis is the potential of a completely open-source research tool for dynamic FEA. Because of the different tools used and the object-oriented programming technique, it can easily be expanded into any research field where new ideas that include large-scale numerical analysis need to be investigated.

It should also be mentioned that the development of these scrips require competence in many different engineering fields. It is important not only to have knowledge about structural mechanics but also in combination with advanced linear algebra and skills in programming. Mastering all of these fields takes time and cannot be done as part of a single master thesis. With that said, still both the programming and research related goal with this thesis was met. When faced with complex programming tasks, the time for refactoring and optimization are endless. It is therefore important to set up a reasonable time frame for such tasks.

In section 6.1, conclusions and some remarks from the present work are presented. Suggestions on further research on the topics that this thesis address are summarized in section 6.2.

6.1 Conclusions

6.1.1 Dynamics of Piles and Pile Groups

The dynamic behavior of piles and pile groups is depending on many factors. Some aspects that are mentioned in the literature are

- plastic deformations of the soil along the pile,
- non-linear material behavior for large strains,
• pile-soil-pile interaction effects because of closely spaces piles and
• vertical and horizontal soil heterogeneity.

To understand the fundamental behavior of pile groups, linear elastic models can still be of use. Numerical methods could be an effective tool to analyze both single piles and pile groups, if proper tools were available. The numerical methods should be used for typical three-dimensional problems such as slanted piles or pile groups with complex geometries. From the case study that has been performed, the following main conclusion can be drawn:

• numerical steady-state analysis can predict the dynamic properties of piles,
• the outcome is dependent on the quality of the mesh and the efficiency of the radiating boundary.

6.1.2 Solutions Methods for Steady-state Analysis

The following conclusion can be drawn about performing FEA and steady-state analysis in parallel:

• Standard iterative methods and preconditioners are well suited for ordinary static FEA where the coefficient matrix is symmetric positive definite.
• Eigenvalue problems are well suited for iterative methods.
• Iterative methods are not suitable for steady-state analysis where the problem is formulated as $K(\omega)u = F$ because the coefficient matrix is complex and indefinite.
• If a linearization is performed on the steady-state problem, resulting in a system with twice as many unknowns, iterative methods may work. Iterative methods has two major benefits compared to direct methods, namely:
  – a costly direct factorization is not needed at each frequency
  – less requirements on available RAM
  – improvements on the required calculation time

The performance of two different scalable direct solvers has been tested. The Distributed SuperLU libraries were found to have the best performance, both concerning time and memory consumption, for this type of problems.
6.2 Future Research

Future research projects that come out of this thesis are divided into several parts. First some general suggestions will be mentioned and later, some more specific programming development and research ideas. The general suggestions are:

- An evaluation on the performance of scalable direct solvers for steady-state analysis on supercomputers. If good performance is found, an iterative solution method could be abandoned for this problem formulation.

- Perform the steady-state analysis on a slanted pile and compare the results with a vertical pile.

- Implement constraints that make it possible to investigate large pile groups and perform a steady-state analysis on them. Here, the frequency range cannot be limited to $a_0 < 0.5$, so this is a major computational task.

- The scope of this thesis has been limited to only include full interaction between the pile and the surrounding soil. A further investigation of how the dynamic stiffness function varies if either interaction elements or when a gap is modeled in the mesh, could improve the knowledge of soil-bridge interaction and maybe explain some of the discrepancies when measurements and FEA results are compared.

- Iterative methods has shown great performance on static FEA. This potential can be used for many purposes, such as:
  - probabilistic design approaches and
  - scripts for solving complicated load combinations.

6.2.1 Script Development

Besides the script development that has been mentioned earlier, some programming oriented project topics are the following:

- code refactoring and optimization,
- subroutines in Fortran for typical sequentially numerical calculations such as integration of element mass and stiffness matrices,
- Lagrange multipliers for constraint equations,
- implementation of other boundary conditions both for dynamic and static analysis,
- implementation of reduced integration for the hexahedral element with hourglass control, and
• to make it possible to use an open-source mesh generator.

The scripts could also be expanded to incorporated other material models, including non-linear behavior. Quadratic continuum element, shell- and/or beam-elements can be implemented. Evidently, the list of possibilities are endless.

6.2.2 Research on the Dynamics of Piles and Pile Groups

A number of different interesting topics are available on the subject of dynamic behavior of piles and pile groups. All of them require large-scale models for three-dimensional dynamic numerical analysis. The following are some suggestions on topics:

• dynamic properties of slanted piles,
• the dynamic influence of closely spaced piles - vertical or slanted,
• the dynamic distribution of loads in pile groups,
• the influence of soil heterogeneity and large deformations on pile behavior, and
• dynamic properties of large pile groups with complex geometries.
References


Appendix A

Mode Shapes

This appendix holds the mode shapes calculated for verification as described in section 4.3. For each mode, the figure calculated with the EPS-script and ABAQUS are shown side-by-side.

Figure A.1: Mode shapes for the first bending mode.
(a) EPS-script (1.0543 Hz), (b) ABAQUS (1.0542 Hz)

Figure A.2: Mode shapes for the first vertical mode.
(a) EPS-script (1.5945 Hz), (b) ABAQUS (1.5943 Hz)
APPENDIX A. MODE SHAPES

Figure A.3: Mode shapes for the first horizontal mode.
(a) EPS-script (2.2270 Hz), (b) ABAQUS (2.2267 Hz)

Figure A.4: Mode shapes for the second horizontal mode.
(a) EPS-script (2.7413 Hz), (b) ABAQUS (2.7411 Hz)

Figure A.5: Mode shapes for the second bending mode.
(a) EPS-script (3.7297 Hz), (b) ABAQUS (3.7286 Hz)
Figure A.6: Mode shapes for the second vertical mode.
(a) EPS-script (4.7476 Hz), (b) ABAQUS (4.7459 Hz)
Appendix B

Source Code

This appendix hold all of the source code for the scripts created as described in chapter 3. They have been ordered of importance and appearance, starting with the three main files, followed by the different classes. Classes are built up by header and source files.

List of Source Codes

1  pre.cpp .................................................. 68
2  solve.cpp .................................................. 74
3  post.cpp .................................................. 79
4  mesh.h .................................................... 82
5  mesh.cpp .................................................. 84
6  load.h ..................................................... 94
7  load.cpp .................................................. 95
8  material.h ............................................... 98
9  material.cpp ........................................... 98
10 element.h ............................................... 99
11 element.cpp ............................................ 101
12 node.h .................................................. 102
13 node.cpp .................................................. 103
14 face.h .................................................. 103
15 face.cpp .................................................. 104
Source Code 1: pre.cpp

```cpp
static char help[] = "Pre\n";

#include <algorithm>
#include <map>
#include <iostream>
#include <fstream>
#include <string>
#include <complex>
#include "petscksp.h"
#include "mesh.h"
#include "matrix.h"
#include "load.h"
#include "material.h"

using std::string;

PetscErrorCode VecSetMapLoad(Vec*, PetscInt , Mesh *, multimap<int,int>, multimap<int,int>,
  double, double, double, double, double);
PetscErrorCode VecSetMapDamp(Vec*, PetscInt , Mesh *, multimap<int,int>, multimap<int,int>,
  double, double, double, double, double);
PetscErrorCode VecOut(Vec ,const char *
);
PetscErrorCode MatOut(Mat ,const char *
);

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc, char **args)
{
    // Variables
    PetscErrorCode ierr;
    PetscMPIInt size, rank;
    PetscBool flg,flgKM,flgF,flgC,flgMass;
    PetscInt nELStart, nELEnd, nActiveDofs, nELDofs, nELTotal, *rowcol;
    PetscScalar *Kel, *Mel;
    Mat K, M;
    Vec F, C;
    //
    int nNode;
    Mesh *pMesh;
    Matrix *pKel;
    Matrix *pMel;
    Material *pMaterial;

```
multimap<int, int> mMapDof, mMapActiveDof;
multimap<int, double> mMapLoad, mMapLoadBody;
multimap<int, int>::iterator mapIt, mapItActive;
multimap<int, double>::iterator mapItLoad;

char s_mshnm[500];
char s_outdr[500];
string adressK, adressM, adressC, adressF;
std::stringstream ss_outdr;

flgKM = PETSC_TRUE;
flgF = PETSC_TRUE;
flgC = PETSC_TRUE;
flgMass = PETSC_FALSE;

PetscInitialize(&argc, &args, (char *)0, help);
ierr = MPI_Comm_size(PETSC_COMM_WORLD, &size); CHKERRQ(ierr);
ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank); CHKERRQ(ierr);

ierr = PetscOptionsGetString(PETSC_NULL, "−flnm", s_mshnm, 500, &flg); CHKERRQ(ierr);
if (!flg) { SETERRQ(PETSC_COMM_WORLD, 1, "Mesh filename is not set! (Option: −flnm < string>)"); }

ierr = PetscOptionsGetString(PETSC_NULL, "−fldr", s_outdr, 500, &flg); CHKERRQ(ierr);
if (!flg) { SETERRQ(PETSC_COMM_WORLD, 1, "Output directory is not set! (Option: −fldr < string>)"); }

ierr = PetscOptionsGetBool(PETSC_NULL, "−KM", &flgKM, PETSC_NULL); CHKERRQ(ierr);
ierr = PetscOptionsGetBool(PETSC_NULL, "−F", &flgF, PETSC_NULL); CHKERRQ(ierr);
ierr = PetscOptionsGetBool(PETSC_NULL, "−C", &flgC, PETSC_NULL); CHKERRQ(ierr);
ierr = PetscOptionsGetBool(PETSC_NULL, "−Mass", &flgMass, PETSC_NULL); CHKERRQ(ierr);

ss_outdr << s_outdr;
adressK = ss_outdr.str() + "K.dat";
adressM = ss_outdr.str() + "M.dat";
adressC = ss_outdr.str() + "C.dat";
adressF = ss_outdr.str() + "F.dat";

MPI_Barrier(PETSC_COMM_WORLD);

PetscPrintf(PETSC_COMM_WORLD, "Load mesh \n"); CHKERRQ(ierr);
pMesh = new Mesh;
pMesh->loadMesh(s_mshnm);
mMapDof = pMesh->getMapMeshDof();
mMapActiveDof = pMesh->mapMeshActiveDof();
nElTotal = pMesh->getElements().size();
nElStart = rank*nElTotal/size;
nElEnd = (1+rank)*nElTotal/size;
nActiveDofs = pMesh->getTotActiveDofs();
if (!nActiveDofs) { SETERRQ(PETSC_COMM_WORLD, 1, "File not found"); } 

MPI_Barrier(PETSC_COMM_WORLD);

PetscPrintf(PETSC_COMM_WORLD, "Create matrix K,M \n"); CHKERRQ(ierr);
ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD, PETSC_DECIDE, PETSC_DECIDE, nActiveDofs, nActiveDofs, 0, PETSC_NULL, 0, PETSC_NULL, &K); CHKERRQ(ierr);
ierr = MatSetFromOptions(K); CHKERRQ(ierr);
APPENDIX B. SOURCE CODE

```c
ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD, PETSC_DECIDE, PETSC_DECIDE, nActiveDofs, nActiveDofs, 0, PETSC_NULL, 0, PETSC_NULL, &M); CHKERRQ(ierr);
ierr = MatSetFromOptions(M); CHKERRQ(ierr);
if (flgKM)
{
    for (int nEl=nElStart; nEl<nElEnd; nEl++)
    {
        nElDofs = 3*(pMesh->getElements().at(nEl)->getElementNumberOfNodes());
ierr = PetscMalloc(nElDofs*nElDofs*sizeof(PetscScalar), &Kel); CHKERRQ(ierr);
ierr = PetscMalloc(nElDofs*nElDofs*sizeof(PetscScalar), &Mel); CHKERRQ(ierr);
ierr = PetscMalloc(nElDofs*sizeof(PetscInt), &rowcol); CHKERRQ(ierr);
pKel = pMesh->getElements().at(nEl)->getElementK();
pMel = pMesh->getElements().at(nEl)->getElementM();
//
int k=0;
for (int i=0; i<nElDofs; i++)
    {
        for (int j=0; j<nElDofs; j++)
            {
                Kel[k] = pKel->getMatValue(i,j);
                Mel[k] = pMel->getMatValue(i,j);
                k++;
            }
    }
delete pKel;
delete pMel;
//
for (int n=0; n<pMesh->getElements().at(nEl)->getElementNumberOfNodes(); n++)
{
    nNode = pMesh->getElements().at(nEl)->getElementTopology().at(n)->
        getNodeNumber();
    int m=0;
    for (mapItActive = mMapActiveDof.equal_range(nNode).first;
        mapItActive != mMapActiveDof.equal_range(nNode).second;
        ++mapItActive)
        {
            if (mapItActive->second != 0)
                rowcol[3*n+m] = mapItActive->second-1;
            else
                rowcol[3*n+m] = -1;
            m++;
        }
ierr = MatSetValues(K, nElDofs, rowcol, nElDofs, rowcol, Kel, ADD_VALUES); CHKERRQ(ierr);
ierr = MatSetValues(M, nElDofs, rowcol, nElDofs, rowcol, Mel, ADD_VALUES); CHKERRQ(ierr);
ierr = PetscFree(rowcol); CHKERRQ(ierr);
ierr = PetscFree(Kel); CHKERRQ(ierr);
ierr = PetscFree(Mel); CHKERRQ(ierr);
}

ierr = PetscPrintf(PETSC_COMM_WORLD,"Assembly begin K "); CHKERRQ(ierr);
ierr = MatAssemblyBegin(K, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(K, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"-end \n Assembly M begin"); CHKERRQ(ierr);
ierr = MatAssemblyBegin(M, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(M, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
```

ierr = PetscPrintf(PETSC_COMM_WORLD,"\n");CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"\n");CHKERRQ(ierr);
ierr = MatOut(K,adressK.c_str());CHKERRQ(ierr);
ierr = MatOut(M,adressM.c_str());CHKERRQ(ierr);
}

// Output TOTAL MASS
if( flgMass ) {
  Vec D;
  PetscScalar sum;
  PetscScalar scale = PetscScalar(0.333333333333,0);
  ierr = VecCreate(PETSC_COMM_WORLD,&D);CHKERRQ(ierr);
  ierr = VecSetSizes(D,PETSC_DECIDE,nActiveDofs);CHKERRQ(ierr);
  ierr = VecSetFromOptions(D);CHKERRQ(ierr);
  ierr = VecSet(D,0.0);CHKERRQ(ierr);
  ierr = MatGetRowSum(M,D);CHKERRQ(ierr);
  ierr = VecScale(D,scale);CHKERRQ(ierr);
  ierr = VecSum(D,&sum);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"MASS %g\n",sum);CHKERRQ(ierr);
  ierr = VecDestroy(&D);CHKERRQ(ierr);
}

// Create and assemble radiating boundary
ierr = PetscPrintf(PETSC_COMM_WORLD,"\n");CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD,&C);CHKERRQ(ierr);
ierr = VecSetSizes(C,PETSC_DECIDE,nActiveDofs);CHKERRQ(ierr);
ierr = VecSetFromOptions(C);CHKERRQ(ierr);
ierr = VecSet(C,0.0);CHKERRQ(ierr);
if( flgC ) {
  ierr = PetscPrintf(PETSC_COMM_WORLD,"\n");CHKERRQ(ierr);
  VecSetMapDamp(&C,nActiveDofs,pMesh,mMapActiveDof, pMesh->getMapElSurfaceSide(),mDamperP,mDamperP,mDamperS,mDamperS,mDamperS);
  ierr = VecAssemblyBegin(C);CHKERRQ(ierr);
  ierr = VecAssemblyEnd(C);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD,"\n");CHKERRQ(ierr);
  ierr = VecOut(C,adressC.c_str());CHKERRQ(ierr);
}

// Create right--hand--side
ierr = PetscPrintf(PETSC_COMM_WORLD,"\n");CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD,&F);CHKERRQ(ierr);
ierr = VecSetSizes(F,PETSC_DECIDE,nActiveDofs);CHKERRQ(ierr);
ierr = VecSetFromOptions(F);CHKERRQ(ierr);
ierr = VecSet(F,0.0);CHKERRQ(ierr);
}

71
// Assemble right-hand-side vector
if (flgF) {
    ierr = PetscPrintf(PETSC_COMM_WORLD, "Assemble RHS \n"); CHKERRQ(ierr);
    if (rank == 0) {
        ierr = VecSetMapLoad(&F, nActiveDofs, pMesh, mMapActiveDof, pMesh->
                         getMapElSurfaceLoad(), 1, 0, 0, 0, 0); CHKERRQ(ierr);
    }
    ierr = VecAssemblyBegin(F); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(F); CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD, "Output F \n"); CHKERRQ(ierr);
    ierr = VecOut(F, adressF.c_str()); CHKERRQ(ierr);
}

scroll

// Finalize and destroy
ierr = PetscPrintf(PETSC_COMM_WORLD, "Petsc finalize \n"); CHKERRQ(ierr);
ierr = MatDestroy(&K); CHKERRQ(ierr);
ierr = MatDestroy(&M); CHKERRQ(ierr);
ierr = VecDestroy(&C); CHKERRQ(ierr);
ierr = VecDestroy(&F); CHKERRQ(ierr);
delete pMesh;
ierr = PetscFinalize();
return 0;

PetscErrorCode VecSetMapLoad(Vec* vec, PetscInt nADof, Mesh* pMesh, multimap<int, int> mMapActiveDof, multimap<int, double> mapSurface, double XX, double YY, double ZZ, double XY, double ZX, double YZ)
{
    Load *pLoad;
    PetscErrorCode ierr;
    PetscScalar *v;
    PetscInt *d;
    ierr = PetscMalloc(nADof*sizeof(PetscScalar),&v); CHKERRQ(ierr);
    ierr = PetscMalloc(nADof*sizeof(PetscInt),&d); CHKERRQ(ierr);
    //
    multimap<int,double> mMapLoad;
    multimap<int,double>::iterator mapItLoad;
    multimap<int,int>::iterator mapItActive;
    //
    pLoad = new SurfaceTraction(&mapSurface,XX,YY,ZZ,XY,ZX,YZ);
    mMapLoad = pLoad->getMeshElNodalLoading(pMesh);
    //
    for (mapItActive = mMapActiveDof.begin(); mapItActive != mMapActiveDof.end(); mapItActive++)
    {
        if (mMapLoad.count(mapItActive->first) > 0)
        {
            mapItLoad = mMapLoad.find(mapItActive->first);
            if (mapItActive->second != 0) 
            {
                v[mapItActive->second-1] = mapItLoad->second;
                d[mapItActive->second-1] = mapItActive->second-1;
            }
            mapItLoad++;
            mapItActive++;
        }
    }
    return 0;
}
if (mapItActive→second !=0) {
    v[mapItActive→second−1] = mapItLoad→second;
    d[mapItActive→second−1] = mapItActive→second−1;
}
    mapItLoad++;
    mapItActive++;
    if (mapItActive→second !=0) {
        v[mapItActive→second−1] = mapItLoad→second;
        d[mapItActive→second−1] = mapItActive→second−1;
    }
} else if (mapItActive→second != 0) {
    v[mapItActive→second−1] = 0;
    d[mapItActive→second−1] = mapItActive→second−1;
}
    ierr = VecSetValues(*vec,nADof,d,v,ADD_VALUES);CHKERRQ(ierr);
    ierr = PetscFree(v);CHKERRQ(ierr);
    ierr = PetscFree(d);CHKERRQ(ierr);
    delete pLoad;
    return 0;
}

PetscErrorCode VecSetMapDamp(Vec* vec, PetscInt nADof, Mesh *pMesh, multimap<int,int> mMapActiveDof, multimap<int,int> mMapSurface,double xN,double yN,double xyS,double yxS,double zS) {
    Load *pLoad;
    PetscErrorCode ierr;
    PetscScalar *v;
    PetscInt *d;
    ierr = PetscMalloc(nADof*sizeof(PetscScalar),&v);CHKERRQ(ierr);
    ierr = PetscMalloc(nADof*sizeof(PetscInt),&d);CHKERRQ(ierr);
    multimap<int,double> mMapLoad;
    multimap<int,double>::iterator mapItLoad;
    multimap<int,int>::iterator mapItActive;
    pLoad = new BoundaryDamping(&mapSurface,xN,yN,xyS,yxS,zS);
    mMapLoad = pLoad->getMeshNodalLoading(pMesh);
    for (mapItActive = mMapActiveDof.begin(); mapItActive != mMapActiveDof.end(); mapItActive++) {
        if (mMapLoad.count(mapItActive→first) > 0) {
            mapItLoad = mMapLoad.find(mapItActive→first);
            if (mapItActive→second !=0) {
                v[mapItActive→second−1] = mapItLoad→second;
                d[mapItActive→second−1] = mapItActive→second−1;
            }
            mapItLoad++;
            mapItActive++;
            if (mapItActive→second !=0) {
                v[mapItActive→second−1] = mapItLoad→second;
            }
        }
    }
}
APPENDIX B. SOURCE CODE

```c

d[mapItActive->second−1] = mapItActive->second−1;
}
mapItLoad++;
mapItActive++;
if (mapItActive->second != 0) {
  v[mapItActive->second−1] = mapItLoad->second;
  d[mapItActive->second−1] = mapItActive->second−1;
}
else if (mapItActive->second != 0) {
  v[mapItActive->second−1] = 0;
  d[mapItActive->second−1] = mapItActive->second−1;
}

// ierr = VecSetValues(∗vec,nADof,d,v,ADD_VALUES);CHKERRQ(ierr);
ierr = PetscFree(v);CHKERRQ(ierr);
ierr = PetscFree(d);CHKERRQ(ierr);
delete pLoad;
return 0;
}

PetscErrorCode VecOut(Vec A, const char ∗ Address)
{
  PetscErrorCode ierr;
  PetscViewer viewer;
  ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,Address,FILE_MODE_WRITE,&viewer);
  ierr = VecView(A,viewer);CHKERRQ(ierr);
  ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);
  return 0;
}

PetscErrorCode MatOut(Mat A, const char ∗ Address)
{
  PetscErrorCode ierr;
  PetscViewer viewer;
  ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,Address,FILE_MODE_WRITE,&viewer);
  ierr = MatView(A,viewer);CHKERRQ(ierr);
  ierr = PetscViewerDestroy(&viewer);CHKERRQ(ierr);
  return 0;
};
```

---

Source Code 2: solve.cpp

```c

static char help[] = "SS-FEA\n";

#include <iostream>
#include <sstream>
#include <string>
#include "petscksp.h"

PetscErrorCode VecOut(Vec, const char ∗);
PetscErrorCode VecIn(Vec, const char ∗);
```

74
PetscErrorCode MatIn(Mat, const char *);
PetscErrorCode KSPOut(KSP, Mat, Vec, Vec, PetscLogDouble, PetscLogDouble, int, const char *);

#define __FUNCTION__ "main"
int main(int argc, char **args)
{
    // Variables
    PetscErrorCode ierr;
    PetscMPIInt size, rank;
    PetscInt nadof, step_start, step_stop;
    PetscScalar Kscale, Mscale, Cscale;
    PetscReal step_freq, freq;
    PetscBool flg;
    PetscLogDouble tall, tsolve, t1, t2, t3;
    // Mat K, M, Kstep;
    Vec F, u, C, Cstep;
    // KSP ksp;
    PC pc;
    // char s_fldr[500];
    std::string adressU, adressK, adressM, adressC, adressF, adressKSP;
    std::stringstream ss_step, ss_fldr;
    // Initialize PETSc and MPI communicator
    ierr = PetscInitialize(&argc, &args, (char *)0, help);
    CHKERRQ(ierr);
    ierr = MPI_Comm_size(PETSC_COMM_WORLD, &size);CHKERRQ(ierr);
    ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank);CHKERRQ(ierr);
    // Run-time options from user
    ierr = PetscOptionsGetInt(PETSC_NULL, "−nadof", &nadof, &flg);CHKERRQ(ierr);
    ierr = PetscOptionsGetInt(PETSC_NULL, "−start", &step_start, &flg);CHKERRQ(ierr);
    ierr = PetscOptionsGetInt(PETSC_NULL, "−stop", &step_stop, &flg);CHKERRQ(ierr);
    ierr = PetscOptionsGetReal(PETSC_NULL, "−step", &step_freq, &flg);CHKERRQ(ierr);
    ierr = PetscOptionsGetReal(PETSC_NULL, "−start", &step_start, &flg);CHKERRQ(ierr);
    ierr = PetscOptionsGetReal(PETSC_NULL, "−stop", &step_stop, &flg);CHKERRQ(ierr);
    ierr = PetscOptionsGetReal(PETSC_NULL, "−step", &step_freq, &flg);CHKERRQ(ierr);

    // Directory is not set! (Option: −fdlr <string>)
    ss_fldr << s_fldr.str();
    adressK = ss_fldr.str() + "K.dat";
    adressM = ss_fldr.str() + "M.dat";
    adressC = ss_fldr.str() + "C.dat";
    adressF = ss_fldr.str() + "F.dat";

    // Load global matrices and vectors
    ierr = PetscPrintf(PETSC_COMM_WORLD, "Load K \n"); CHKERRQ(ierr);
APPENDIX B. SOURCE CODE

```c
ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,nadof,
    nadof,0,PETSC_NULL,0,PETSC_NULL,&K);CHKERRQ(ierr);
ierr = MatSetFromOptions(K);CHKERRQ(ierr);
ierr = MatIn(K,adressK.c_str());

/* Load M */
CHKERRQ(ierr);
ierr = MatCreateMPIAIJ(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,nadof,
    nadof,0,PETSC_NULL,0,PETSC_NULL,&M);CHKERRQ(ierr);
ierr = MatSetFromOptions(M);CHKERRQ(ierr);
ierr = MatIn(M,adressM.c_str());

/* Load Cr */
ierr = VecCreate(PETSC_COMM_WORLD,&C);CHKERRQ(ierr);
ierr = VecSetFromOptions(C);CHKERRQ(ierr);
ierr = VecIn(C,adressC.c_str());

/* Load RHS (F) */
ierr = VecCreate(PETSC_COMM_WORLD,&F);CHKERRQ(ierr);
ierr = VecSetFromOptions(F);CHKERRQ(ierr);
ierr = VecIn(F,adressF.c_str());

// Create solution vectors
ierr = VecCreate(PETSC_COMM_WORLD,&u);CHKERRQ(ierr);
ierr = VecSetSizes(u,PETSC_DECIDE,nadof);CHKERRQ(ierr);
ierr = VecSetFromOptions(u);CHKERRQ(ierr);
ierr = VecSet(u,0.0);CHKERRQ(ierr);

// Create KSP
ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);

// Scale K for rate-independent damping
Kscale = PetscScalar(1.0,0.04);
ierr = MatScale(K,Kscale);CHKERRQ(ierr);

// Solve KSP start
for (int step=step_start; step<=(step_start+(step_stop-step_start)/step_freq); step++)
{
    ierr = PetscGetTime(&t1);CHKERRQ(ierr);

    freq = step_start+(step-step_start)*step_freq;
    u_step = freq;
    adressU = ss_fldr.str()+"u_"+ss_step.str()+".dat";
    adressKSP = ss_fldr.str()+"KSP_"+ss_step.str()+".dat";
    ss_step.str(" ");

    // Scale for current step
    Mscale = PetscScalar(-39.4784176*freq*freq,0);
    Cscale = PetscScalar(0,6.28318531*freq);

    ierr = MatDuplicate(K,MAT_COPY_VALUES,&Kstep);CHKERRQ(ierr);
ierr = MatAXPY(Kstep,Mscale,M,SAME_NONZERO_PATTERN);CHKERRQ(ierr);
ierr = VecDuplicate(C,&Cstep);CHKERRQ(ierr);
ierr = VecCopy(C,Cstep);CHKERRQ(ierr);
ierr = VecCopy(C,Cstep);CHKERRQ(ierr);
```

ierr = VecScale(Cstep,Cscale);CHKERRQ(ierr);
ierr = MatDiagonalSet(Kstep,Cstep,ADD_VALUES);CHKERRQ(ierr);

//*****************************************************************************
ierr = KSPSetOperators(ksp,Kstep,Kstep,Different_NONZERO_PATTERN);CHKERRQ(ierr);
ierr = KSPSetPC(ksp,&pc);CHKERRQ(ierr);
ierr = KSPGetFromOptions(ksp);CHKERRQ(ierr);
ierr = KSPSetUp(ksp);CHKERRQ(ierr);
ierr = PetscGetTime(&t2);CHKERRQ(ierr);
ierr = KSPSolve(ksp,F,u);CHKERRQ(ierr);
ierr = PetscGetTime(&t3);CHKERRQ(ierr);
tsolve = t3 - t2;
tall = t3 - t1;
ierr = PetscPrintf(PETSC_COMM_WORLD,"KSP done 
");CHKERRQ(ierr);

// Store KSP data
ierr = PetscPrintf(PETSC_COMM_WORLD,"Store KSP data 
");CHKERRQ(ierr);
ierr = KSPOut(ksp,Kstep,F,u,tall,tsolve,freq,adressKSP.c_str());CHKERRQ(ierr);

// Store results
ierr = PetscPrintf(PETSC_COMM_WORLD,"Store u 
");CHKERRQ(ierr);
ierr = VecOut(u,adressU.c_str());CHKERRQ(ierr);

// Destroy
ierr = VecDestroy(&Cstep);CHKERRQ(ierr);
ierr = MatDestroy(&Kstep);CHKERRQ(ierr);

MPI_Barrier(PETSC_COMM_WORLD);

// Finalize and destroy
ierr = PetscPrintf(PETSC_COMM_WORLD,"Petsc finalize 
");CHKERRQ(ierr);
ierr = KSPDestroy(&ksp);CHKERRQ(ierr);
ierr = MatDestroy(&K);CHKERRQ(ierr);
ierr = MatDestroy(&M);CHKERRQ(ierr);
ierr = VecDestroy(&F);CHKERRQ(ierr);
ierr = VecDestroy(&u);CHKERRQ(ierr);
ierr = VecDestroy(&C);CHKERRQ(ierr);
ierr = PetscFinalize();
return 0;

PetscErrorCode VecOut(Vec A, const char *Adress)
{
    PetscErrorCode ierr;
    PetscViewer viewer;
    ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,Adress,FILE_MODE_WRITE,&viewer);
    ierr = VecView(A,viewer);CHKERRQ(ierr);
    ierr = VecOut(A,Adress);CHKERRQ(ierr);
    return 0;
}

PetscErrorCode VecIn(Vec A, const char *Adress)
{
    PetscErrorCode ierr;
    PetscViewer viewer;
    ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,Adress,FILE_MODE_READ,&viewer);
    ierr = VecView(A,viewer);CHKERRQ(ierr);
    ierr = VecIn(A,Adress);CHKERRQ(ierr);
    return 0;
}
APPENDIX B. SOURCE CODE

PetscErrorCode MatIn(Mat A, const char *Address)
{
  PetscErrorCode ierr;
  PetscViewer viewer;
  ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD, Address, FILE_MODE_READ, &viewer);
  CHKERRQ(ierr);
  ierr = MatLoad(A, viewer); CHKERRQ(ierr);
  ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
  return 0;
}

PetscErrorCode MatIn(Mat A, const char *Address)
{
  PetscErrorCode ierr;
  PetscViewer viewer;
  ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD, Address, FILE_MODE_READ, &viewer);
  CHKERRQ(ierr);
  ierr = VecLoad(A, viewer); CHKERRQ(ierr);
  ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
  return 0;
}

PetscErrorCode KSPOut(KSP ksp, Mat K, Vec F, Vec u, PetscLogDouble t1, PetscLogDouble t2, int step, const char *Address)
{
  PetscErrorCode ierr;
  PetscViewer viewer;
  KSPConvergedReason reason;
  PetscReal pcnorm, tnorm, RHSnorm;
  PetscInt its;
  Vec x;
  //
  ierr = VecDuplicate(u, &x); CHKERRQ(ierr);
  ierr = MatMult(K, u, x); CHKERRQ(ierr);
  ierr = VecAXPY(x, -1.0, F); CHKERRQ(ierr);
  ierr = VecNorm(x, NORM_2, &tnorm); CHKERRQ(ierr);
  ierr = VecNorm(F, NORM_2, &RHSnorm); CHKERRQ(ierr);
  //
  ierr = KSPGetIterationNumber(ksp, &its); CHKERRQ(ierr);
  ierr = KSPGetConvergedReason(ksp, &reason); CHKERRQ(ierr);
  ierr = KSPGetResidualNorm(ksp, &pcnorm); CHKERRQ(ierr);
  //
  ierr = PetscViewerASCIIOpen(PETSC_COMM_WORLD, Address, &viewer); CHKERRQ(ierr);
  ierr = PetscViewerASCIIPrintf(viewer, "−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−


"); CHKERRQ(ierr);
  ierr = PetscViewerASCIIPrintf(viewer, " K S P S o l v e 


"); CHKERRQ(ierr);
  ierr = PetscViewerASCIIPrintf(viewer, " Step: %d
", step); CHKERRQ(ierr);
  ierr = PetscViewerASCIIPrintf(viewer, " Total time: (s) %1.4e
", t1); CHKERRQ(ierr);
  ierr = PetscViewerASCIIPrintf(viewer, " Total time for solve: (s) %1.4e
", t2); CHKERRQ(ierr);
  if (reason<0){ ierr = PetscViewerASCIIPrintf(viewer, " D I V E R G E D !

"); CHKERRQ(ierr);
  };
  if (reason>=0){ ierr = PetscViewerASCIIPrintf(viewer, " C O N V E R G E D !

"); CHKERRQ(ierr);
  };
  if (reason==0){ ierr = PetscViewerASCIIPrintf(viewer, " K S P _ C O N V E R G E D _ I T E R A T I N G (%d)
", reason); CHKERRQ(ierr); };
  if (reason==1){ ierr = PetscViewerASCIIPrintf(viewer, " K S P _ C O N V E R G E D _ R T O L _ N O R M A L (%d)
", reason); CHKERRQ(ierr); };
  if (reason==9){ ierr = PetscViewerASCIIPrintf(viewer, " K S P _ C O N V E R G E D _ A T O L _ N O R M A L (%d)
", reason); CHKERRQ(ierr); };
  return ierr;
}
if (reason==2){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_CONVERGED_RTOL (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==3){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_CONVERGED_ATOL (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==4){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_CONVERGED_ITS (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==5){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_CONVERGED_CG_NEG_CURVE (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==6){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_CONVERGED_CG_CONSTRAINED (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==7){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_CONVERGED_STEP_LENGTH (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==8){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_CONVERGED_HAPPY_BREAKDOWN (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−2){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_NULL (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−3){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_ITS (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−4){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_DTOL (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−5){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_BREAKDOWN (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−6){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_BREAKDOWN_BICG (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−7){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_NONSYMMETRIC (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−8){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_INDEFINITE_PC (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−9){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_NAN (%d)\n\n",reason);CHKERRQ(ierr);};
if (reason==−10){ierr = PetscViewerASCIIPrintf(viewer, " Reason: KSP_DIVERGED_INDEFINITE_MAT (%d)\n\n",reason);CHKERRQ(ierr);};
//
#include <map>
#include <iostream>
#include <fstream>
#include <string>

Source Code 3: post.cpp

static char help[] = "Post\n";

#include <map>
#include <iostream>
#include <fstream>
#include <string>
#include <complex>
#include "petscmat.h"
#include "mesh.h"

using std::string;

PetscErrorCode VecIn(Vec, const char ∗);

#undef __FUNCT__
#define __FUNCT__ "main"

int main(int argc, char ∗∗args)
{

    // Variables
    PetscErrorCode ierr;
    PetscMPIInt size;
    PetscInt ndof, stepStart, stepStop;
    PetscReal stepRange, freq;
    PetscScalar ∗uN, ∗FN, ∗CN;
    PetscBool flg;
    Vec u, F, C;
    PetscBool storeF = PETSC_TRUE;
    PetscBool storeC = PETSC_TRUE;
    Mesh ∗pMesh;
    multimap<int, int> mapDof, mapADof;
    multimap<int, int>::iterator mapIt, mapItA;
    char sMshnm[500], sFldr[500];
    string adressU, adressResults, adressF, adressC, adressNodeReal, adressNodeImag;
    std::stringstream ssStep, ssStep2, ssFldr;

    // Initialize PETSc
    PetscInitialize(&argc, &args, (char *)0, help);
    ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
    if (size>1){ SETERRQ(PETSC_COMM_WORLD,1,"Only for one processor"); }
    ierr = PetscOptionsGetInt(PETSC_NULL,"−start", &stepStart,&flg);CHKERRQ(ierr);
    if (!flg){ SETERRQ(PETSC_COMM_WORLD,1,"Step start is not set! (Option: −start <>)"); }
    ierr = PetscOptionsGetInt(PETSC_NULL,"−stop", &stepStop,&flg);CHKERRQ(ierr);
    if (!flg){ SETERRQ(PETSC_COMM_WORLD,1,"Step stop is not set! (Option: −stop <>)"); }
    ierr = PetscOptionsGetReal(PETSC_NULL,"−step", &stepRange,&flg);CHKERRQ(ierr);
    if (!flg){ stepRange = 1; }
    if (stepStart>=stepStop){ SETERRQ(PETSC_COMM_WORLD,1,"Stop needs to be larger then start"); }
    ierr = PetscOptionsGetString(PETSC_NULL,"−flnm", sMshnm,500,&flg);CHKERRQ(ierr);
    if (!flg){ SETERRQ(PETSC_COMM_WORLD,1,"Mesh filename is not set! (Option: −flnm <string>)" ); }
    ierr = PetscOptionsGetString(PETSC_NULL,"−fldr", sFldr,500,&flg);CHKERRQ(ierr);
    if (!flg){ SETERRQ(PETSC_COMM_WORLD,1,"Directory is not set! (Option: −fldr <string>)" ); }
    ssFldr << sFldr;
    adressF = ssFldr.str()+"F.dat";
    adressC = ssFldr.str()+"C.dat";

    // Create mesh and map dof, adof and load
    ierr = PetscPrintf(PETSC_COMM_WORLD,"Load mesh \n");CHKERRQ(ierr);
    pMesh = new Mesh;

}
pMesh->loadMesh(sMshnm);
mapDof = pMesh->getMapMeshDof();
mapADof = pMesh->mapMeshActiveDof();
ndof = mapADof.size();
vector<double> fOut(ndof,0.0);
vector<double> cOut(ndof,0.0);
vector<double> uReal;
vector<double> uImag;

//
for (int step=stepStart; step<=(stepStart+(stepStop−stepStart)/stepRange); step++)
{
uReal = vector<double>(ndof,0.0);
uImag = vector<double>(ndof,0.0);
freq = stepStart+(step−stepStart)*stepRange;
ssStep << freq;
ssStep2 << step;
adressU = ssFldr.str()+"u_"+ssStep.str()+".dat";
adressResults = ssFldr.str()+"res_range_"+ssStep2.str()+".vtu";
ssStep.str(" ");
ssStep2.str(" ");
ierr = PetscPrintf(PETSC_COMM_WORLD,"Load u");CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD,&u);CHKERRQ(ierr);
ierr = VecSetFromOptions(u);CHKERRQ(ierr);
ierr = VecIn(u,adressU.c_str());CHKERRQ(ierr);
ierr = VecIn(u,adressU.c_str());CHKERRQ(ierr);
mapIt = mapDof.begin();
for (mapItA = mapADof.begin();mapItA != mapADof.end();mapItA++)
{
if (mapItA−>second != 0)
{
uReal.at(mapItA−>second−1) = uN[mapItA−>second−1].real();
uImag.at(mapItA−>second−1) = uN[mapItA−>second−1].imag();
}
mapIt++;
}
ierr = VecRestoreArray(u,&uN);CHKERRQ(ierr);
if (storeF) // only once!
{
ierr = PetscPrintf(PETSC_COMM_WORLD,"Load F");CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD,&F);CHKERRQ(ierr);
ierr = VecSetFromOptions(F);CHKERRQ(ierr);
ierr = VecIn(F,adressF.c_str());CHKERRQ(ierr);
ierr = VecGetArray(F,&FN);CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,"→store F\n");CHKERRQ(ierr);
mapIt = mapDof.begin();
for (mapItA = mapADof.begin();mapItA != mapADof.end();mapItA++)
{
if (mapItA−>second != 0)
{
fOut.at(mapItA−>second−1) = FN[mapItA−>second−1].real();
}
mapIt++;
}
storeF = PETSC_FALSE;
ierr = VecRestoreArray(F,&FN); CHKERRQ(ierr);
APPENDIX B. SOURCE CODE

```c
ierr = VecDestroy(&F); CHKERRQ(ierr);
}
if (storeC) // only once!
{
ierr = PetscPrintf(PETSC_COMM_WORLD,"Load C"); CHKERRQ(ierr);
ierr = VecCreate(PETSC_COMM_WORLD,&C); CHKERRQ(ierr);
ierr = VecSetFromOptions(C); CHKERRQ(ierr);
ierr = VecIn(C,adressC.c_str()); CHKERRQ(ierr);
ierr = PetscPrintf(PETSC_COMM_WORLD,">store C
"); CHKERRQ(ierr);
mapIt = mapDof.begin();
for (mapItA = mapADof.begin();mapItA != mapADof.end();mapItA++)
{
    if (mapItA->second != 0)
    {
        cOut.at(mapItA->second-1) = CN[mapItA->second-1].real();
    }
    mapItA++;
} 
storeC = PETSC_FALSE;
ierr = VecRestoreArray(C,&CN); CHKERRQ(ierr);
ierr = VecDestroy(&C); CHKERRQ(ierr);
}
ierr = PetscPrintf(PETSC_COMM_WORLD,">sendMesh
"); CHKERRQ(ierr);
pMesh->sendMesh(adressResults,uReal,uImag,fOut,cOut);
uReal.clear();
uImag.clear();
ierr = VecDestroy(&u); CHKERRQ(ierr);
}
#endif
```

Source Code 4: mesh.h

```c
#ifndef MESH_H
#define MESH_H
#include <iostream>
#include <fstream>

PetscErrorCode VecIn(Vec A, const char * Address)
{
    PetscErrorCode ierr;
PetscViewer viewer;
ierr = PetscViewerBinaryOpen(PETSC_COMM_WORLD,Address,FILE_MODE_READ,&viewer);
    CHKERRQ(ierr);
ierr = VecLoad(A,viewer); CHKERRQ(ierr);
ierr = PetscViewerDestroy(&viewer); CHKERRQ(ierr);
    return 0;
};
```

Source Code 4: mesh.h

```c
#ifndef MESH_H
#define MESH_H
#include <iostream>
#include <fstream>

```

82
#include <sstream>
#include <string>
#include <vector>
#include <map>
#include <set>

#include "node.h"
#include "C3D8.h"
#include "C3D4.h"
#include "element.h"
#include "material.h"
#include "matrix.h"
#include "face.h"

using std::string;
using std::vector;
using std::multimap;
using std::set;
using std::pair;

class Mesh
{
public:

    Mesh();
    ~Mesh();

    void loadMesh(string);
    void sendMesh(string,vector<double>,vector<double>,vector<double>,vector<double>);

    vector<Node *> getNodes();
    vector<Element *> getElements();

    multimap<int,int> getMapMeshDof();
    multimap<int,int> mapMeshActiveDof();

    int getTotActiveDofs();

    multimap<int,int> getMapElSurfaceLoad();
    multimap<int,int> getMapElSurfaceSide();

private:

    void newFaceFourNodes(int,int,int,int);
    void newFaceThreeNodes(int,int,int);

    int mNodeNumber;
    int mElementNumber;
    int mElementNumberStart;
    int mElementNumberStop;
    int mNode;
    int mFaceNumber;
    int mStep;

    double mCoordinateX;
    double mCoordinateY;
};
APPENDIX B. SOURCE CODE

double mCoordinateZ;

char skip;
int position;
int mActiveDofs;
string mLine;
string mElementType;
string mSetType;
string mElementFace;

vector<Node *> mNodes;
vector<Face *> mFaces;
vector<Node *> mElementTop;
vector<Face *> mElementFaces;
vector<Node *> mFaceNodes;
vector<Element *> mElements;

multimap<int,int> mMapMeshDof;
multimap<int,int> mMapMeshActiveDof;
multimap<int,int> mMapMeshLoadSet;
multimap<int,int> mMapElSurfaceDomainEdge;
multimap<int,int> mMapElSurfaceLoad;

multimap<int,int>::iterator mapIt;
multimap<int,int>::iterator mapIt2;

set<int> mSetLockedXYZ;
set<int>::iterator setIt;

Material * pSteel;
Material * pConcrete;
Material * pSoil;

}

#endif

Source Code 5: mesh.cpp

#include "mesh.h"

Mesh::Mesh()
{
    pSteel = new Steel();
pConcrete = new Concrete();
pSoil = new Soil();
}

Mesh::~Mesh()
{
    vector<Node *>::iterator itN;
    for( itN=mNodes.begin(); itN<mNodes.end(); itN++){ delete *itN; }
    vector<Face *>::iterator itF;
    for( itF=mFaces.begin(); itF<mFaces.end(); itF++){ delete *itF; }
    vector<Element *>::iterator itE;
    for( itE=mElements.begin(); itE<mElements.end(); itE++){ delete *itE; }

84
void Mesh::loadMesh(string fileName)
{
    std::ifstream inputFile(fileName.c_str(), std::ios::in);
    //
    if (inputFile.is_open())
    {
        while (getline( inputFile, mLine, '\n' ) )
        {
            if ( mLine.compare( 0, 1, "*" ) == 0 )
            {
                if ( mLine.compare( 1, 4, "Node" ) == 0 )
                {
                    position = inputFile.tellg();
                    while( true )
                    {
                        getline( inputFile, mLine, '\n' );
                        if ( mLine.compare( 0, 1, "*" ) == 0 ){ break; } //
                        std::istringstream mLineStream( mLine );
                        mLineStream >> mNodeNumber
                        >> skip >> mCoordinateX
                        >> skip >> mCoordinateY
                        >> skip >> mCoordinateZ;
                        mNodes.push_back( new Node(mNodeNumber, mCoordinateX, mCoordinateY,
                        mCoordinateZ));
                    }
                    inputFile.clear();
                    inputFile.seekg(position, std::ios::beg);
                }
                else if(mLine.compare(1,7,"Element")==0)
                {
                    mFaceNumber = 0;
                    position = inputFile.tellg();
                    mElementType = mLine.substr(15);
                    while(true)
                    {
                        getline(inputFile,mLine, '\n');
                        if(mLine.compare(0,1,"*")==0){ break; }
                        std::istringstream mLineStream( mLine );
                        mLineStream >> mElementNumber;
                        if(mElementType.compare(0,5,"C3D8R")==0)
                        {
                            for(int n=0;n<8;n++)
                            {
                                mLineStream >> skip >> mNode;
                                mElementTop.push_back(mNodes.at(mNode-1));
                            }
                            newFaceFourNodes(0,1,2,3);
                        }
                    }
                }
            }
        }
    }
}
newFaceFourNodes(4,5,6,7);
newFaceFourNodes(0,1,5,4);
newFaceFourNodes(1,2,6,5);
newFaceFourNodes(2,3,7,6);
newFaceFourNodes(3,0,4,7);

//
mElements.push_back(new C3D8(mElementNumber,mElementTop,mElementFaces, pSoil));
//
for (int n=0; n<8; n++)
{
if(mMapMeshDof.count(mElementTop.at(n)->getNodeNumber())==0)
{
    mMapMeshDof.insert(pair<int,int>(mElementTop.at(n)->getNodeNumber(),
        3*(mElementTop.at(n)->getNodeNumber())-2));
    mMapMeshDof.insert(pair<int,int>(mElementTop.at(n)->getNodeNumber(),
        3*(mElementTop.at(n)->getNodeNumber())-1));
    mMapMeshDof.insert(pair<int,int>(mElementTop.at(n)->getNodeNumber()));
}
}
else if (mElementType.compare(0,6,"C3D20R")==0)
{
    for (int n=0; n<20; n++)
    {
        mLineStream >> skip >> mNode;
        mElementTop.push_back(mNodes.at(mNode-1));
    }
    mElements.push_back(new C3D20(mElementNumber,mElementTop,mElementFaces, pSoil));
}
else if (mElementType.compare(0,4,"C3D4")==0)
{
    for (int n=0; n<4; n++)
    {
        mLineStream >> skip >> mNode;
        mElementTop.push_back(mNodes.at(mNode-1));
    }
    newFaceThreeNodes(0,1,2);
    newFaceThreeNodes(0,1,3);
    newFaceThreeNodes(1,2,3);
    newFaceThreeNodes(2,0,3);
    mElements.push_back(new C3D4(mElementNumber,mElementTop,mElementFaces, pSoil));
}
else if (mMapMeshDof.count(mElementTop.at(n)->getNodeNumber())==0)
{
    mMapMeshDof.insert(pair<int,int>(mElementTop.at(n)->getNodeNumber(),
        3*(mElementTop.at(n)->getNodeNumber())-2));
    mMapMeshDof.insert(pair<int,int>(mElementTop.at(n)->getNodeNumber(),
        3*(mElementTop.at(n)->getNodeNumber())-1));
    mMapMeshDof.insert(pair<int,int>(mElementTop.at(n)->getNodeNumber()));
}
}
mMapMeshDof.insert(pair<int,int>(mElementTop.at(n)−getNodeNumber(),(3∗(mElementTop.at(n)−getNodeNumber()))));
}
}
else if (mElementType.compare(0,6,"C3D10M") == 0)
{
    for (int n=0; n<10; n++)
    {
        mLineStream >> skip >> mNode;
        mElementTop.push_back ( mNodes.at( mNode − 1 ) );
    }
    mElements.push_back(new C3D10(mElementNumber,mElementTop,mElementFaces,
pSoil));
}
else
{
    std::cout << "Unknown element!" << std::endl;
    mElementTop.clear();
    mElementFaces.clear();
}
inputFromFile.clear();
inputFromFile.seekg(position, std::ios::beg);
}
else if (mLine.compare(1,4,"Nset") == 0)
{
    position = inputFromFile.tellg();
    mSetType = mLine.substr(12);
    while (true)
    {
        getline(inputFromFile,mLine,'\n');
        if (mLine.compare(0,1,"*" ) == 0)
        { break; }
        std::istringstream mLineStream(mLine);
        if (mSetType.compare(0,6,"LockedXYZ") == 0)
        {
            while (mLineStream.good())
            {
                mLineStream >> mNodeNumber >> skip;
                if ( mSetLockedXYZ.find(mNodeNumber) == mSetLockedXYZ.end() )
                    mSetLockedXYZ.insert( mNodeNumber );
            }
        }
        else
        {
            std::cout << "Unknown set! (Nset) " " << mSetType << std::endl;
            break;
        }
    }
    inputFromFile.clear();
    inputFromFile.seekg(position, std::ios::beg);
}
else if (mLine.compare(1,5,"Elset") == 0)
APPENDIX B. SOURCE CODE

```cpp
{
    position = inputfromfile.tellg();
    mSetType = mLine.substr(14);
    while (true)
    {
        getline(inputfromfile,mLine,'\n');
        if (mLine.compare(0,1,"*") == 0)
        { break; }
    }
    std::istringstream mLineStream(mLine);
    if (mSetType.compare(0,9,"_DomainEdge_S1") == 0)
    {
        while (mLineStream.good())
        {
            mLineStream >> mElementNumber >> skip;
            if ( mMapElSurfaceDomainEdge.find(mElementNumber) ==
                mMapElSurfaceDomainEdge.end() )
                mMapElSurfaceDomainEdge.insert(pair<int,int>( mElementNumber, 1));
        }
    }
    else if (mSetType.compare(0,9,"_DomainEdge_S2") == 0)
    {
        while (mLineStream.good())
        {
            mLineStream >> mElementNumber >> skip;
            if ( mMapElSurfaceDomainEdge.find(mElementNumber) ==
                mMapElSurfaceDomainEdge.end() )
                mMapElSurfaceDomainEdge.insert(pair<int,int>( mElementNumber, 2));
        }
    }
    else if (mSetType.compare(0,9,"_DomainEdge_S3") == 0)
    {
        while (mLineStream.good())
        {
            mLineStream >> mElementNumber >> skip;
            if ( mMapElSurfaceDomainEdge.find(mElementNumber) ==
                mMapElSurfaceDomainEdge.end() )
                mMapElSurfaceDomainEdge.insert(pair<int,int>( mElementNumber, 3));
        }
    }
    else if (mSetType.compare(0,9,"_DomainEdge_S4") == 0)
    {
        while (mLineStream.good())
        {
            mLineStream >> mElementNumber >> skip;
            if ( mMapElSurfaceDomainEdge.find(mElementNumber) ==
                mMapElSurfaceDomainEdge.end() )
                mMapElSurfaceDomainEdge.insert(pair<int,int>( mElementNumber, 4));
        }
    }
    else if (mSetType.compare(0,9,"_DomainEdge_S5") == 0)
    {
        while (mLineStream.good())
        {
            mLineStream >> mElementNumber >> skip;
            if ( mMapElSurfaceDomainEdge.find(mElementNumber) ==
                mMapElSurfaceDomainEdge.end() )
                mMapElSurfaceDomainEdge.insert(pair<int,int>( mElementNumber, 5));
        }
    }
}
```
mLineStream >> mElementNumber >> skip;
if ( mMapElSurfaceDomainEdge.find(mElementNumber) == 
    mMapElSurfaceDomainEdge.end() ) 
    mMapElSurfaceDomainEdge.insert(pair<int,int>( mElementNumber, 5));
}
else if (mSetType.compare(0,9, "_DomainEdge_S6") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceDomainEdge.find(mElementNumber) == 
            mMapElSurfaceDomainEdge.end() ) 
            mMapElSurfaceDomainEdge.insert(pair<int,int>( mElementNumber, 6));
    }
} 
else if (mSetType.compare(0,9, "_Load_S1") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() ) 
            mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 1));
    }
}
else if (mSetType.compare(0,9, "_Load_S2") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() ) 
            mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 2));
    }
}
else if (mSetType.compare(0,9, "_Load_S3") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() ) 
            mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 3));
    }
}
else if (mSetType.compare(0,9, "_Load_S4") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() ) 
            mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 4));
    }
}
else if (mSetType.compare(0,9, "_Load_S5") == 0)
{
    while (mLineStream.good())
    {  

89
mLineStream >> mElementNumber >> skip;
    if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() )
        mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 5));
    
}  
else if (mSetType.compare(0,9,"_Load_S6") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() )
            mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 6));
    }
}
else if (mSetType.compare(0,18,"Concrete, generate") == 0)
{
    mLineStream >> mElementNumberStart >> skip
        >> mElementNumberStop;
    for (int n=mElementNumberStart; n<=mElementNumberStop; n++)
    {
        mElements.at(n−1)→setElementMaterial(pConcrete);
    }
    else
    {
        std::cout << "Unknown set! (Elset) " " << mSetType " " std::endl;
        break;
    }
}  
inputfromfile.clear();
inputfromfile.seekg( position, std::ios::beg );
}
else
{
    std::cout << "Unknown error during stream! " " << std::endl;
}
}  
inputfromfile.close();
}

void Mesh::sendMesh(string fileName, vector<double> mvDefReal, vector<double> mvDefImag,
        vector<double> mvLoad, vector<double> mvDamp )
{
    std::ofstream fileOut(fileName.c_str());

    fileOut << "<?xml version="1.0"?>" " << std::endl
    << "<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">" "
    "<UnstructuredGrid>" " << std::endl
    "<Piece NumberOfPoints=" << mNodes.size()" " << std::endl
    "<PointData>" " << std::endl
    "<UnstructuredGrid>" " << std::endl
    "<Piece NumberOfPoints=" << mNodes.size()" " << std::endl
    "<PointData>" " << std::endl

    mLineStream >> mElementNumber >> skip;
    if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() )
        mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 5));
    
}  
else if (mSetType.compare(0,9,"_Load_S6") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() )
            mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 6));
    }
}
else if (mSetType.compare(0,18,"Concrete, generate") == 0)
{
    mLineStream >> mElementNumberStart >> skip
        >> mElementNumberStop;
    for (int n=mElementNumberStart; n<=mElementNumberStop; n++)
    {
        mElements.at(n−1)→setElementMaterial(pConcrete);
    }
    else
    {
        std::cout << "Unknown set! (Elset) " " << mSetType " " std::endl;
        break;
    }
}  
inputfromfile.clear();
inputfromfile.seekg( position, std::ios::beg );
}
else
{
    std::cout << "Unknown error during stream! " " << std::endl;
}
}  
inputfromfile.close();
}

void Mesh::sendMesh(string fileName, vector<double> mvDefReal, vector<double> mvDefImag,
        vector<double> mvLoad, vector<double> mvDamp )
{
    std::ofstream fileOut(fileName.c_str());

    fileOut << "<?xml version="1.0"?>" " << std::endl
    << "<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">" "
    "<UnstructuredGrid>" " << std::endl
    "<Piece NumberOfPoints=" << mNodes.size()" " << std::endl
    "<PointData>" " << std::endl
    "<UnstructuredGrid>" " << std::endl
    "<Piece NumberOfPoints=" << mNodes.size()" " << std::endl
    "<PointData>" " << std::endl

    mLineStream >> mElementNumber >> skip;
    if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() )
        mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 5));
    
}  
else if (mSetType.compare(0,9,"_Load_S6") == 0)
{
    while (mLineStream.good())
    {
        mLineStream >> mElementNumber >> skip;
        if ( mMapElSurfaceLoad.find(mElementNumber) == mMapElSurfaceLoad.end() )
            mMapElSurfaceLoad.insert(pair<int,int>( mElementNumber, 6));
    }
}
else if (mSetType.compare(0,18,"Concrete, generate") == 0)
{
    mLineStream >> mElementNumberStart >> skip
        >> mElementNumberStop;
    for (int n=mElementNumberStart; n<=mElementNumberStop; n++)
    {
        mElements.at(n−1)→setElementMaterial(pConcrete);
    }
    else
    {
        std::cout << "Unknown set! (Elset) " " << mSetType " " std::endl;
        break;
    }
}  
inputfromfile.clear();
inputfromfile.seekg( position, std::ios::beg );
}
else
{
    std::cout << "Unknown error during stream! " " << std::endl;
}
}  
inputfromfile.close();
}
for (unsigned int n=0; n<mvDefReal.size(); n++) { fileOut << mvDefReal.at(n) << " "; }
for (unsigned int n=0; n<mvDefImag.size(); n++) { fileOut << mvDefImag.at(n) << " "; }
for (unsigned int n=0; n<mvLoad.size(); n++) { fileOut << mvLoad.at(n) << " "; }
for (unsigned int n=0; n<mvDamp.size(); n++) { fileOut << mvDamp.at(n) << " "; }
for (unsigned int n=0; n<mNodes.size(); n++) {
    fileOut << mNodes.at(n)->getNodeX() << " " << mNodes.at(n)->getNodeY() << " " << mNodes.at(n)->getNodeZ() << " ";
}
for (unsigned int n=0; n<mElements.size(); n++) {
    for(unsigned int i=1;i<=mElements.at(n)->getElementTopology().size();i++) {
        fileOut << mElements.at(n)->getElementNodes(i)-1 << " ";
    }
}
for(unsigned int n=0;n<mElements.size();n++) {
    for(unsigned int i=1;i<=mElements.at(n)->getElementTopology().size();i++) {
        fileOut << mElements.at(n)->getElementNodes(i)-1 << " ";
    }
}
APPENDIX B. SOURCE CODE

```cpp
<< " <DataArray type="Int32" Name="offsets" format="ascii">" << std::endl;

int offset = 0;
for(unsigned int n=0;n<mElements.size();n++)
{
    offset = offset + mElements.at(n)->getElementNumberOfNodes();
    fileOut << offset << " ";
}
fileOut << std::endl << " </DataArray>" << std::endl;
<< " <DataArray type="UInt8" Name="types" format="ascii">" << std::endl;
for (unsigned int n=0; n<mElements.size(); n++)
{
    if (mElements.at(n)->getElementNumberOfNodes() == 4)
    {
        fileOut << "10 "; // VTK_TETRA (=10)
    }
    else if (mElements.at(n)->getElementNumberOfNodes() == 8)
    {
        fileOut << "12 "; // VTK_HEXAHEDRON (=12)
    }
    else if (mElements.at(n)->getElementNumberOfNodes() == 10)
    {
        fileOut << "24 "; // VTK_QUADRATIC_TETRA (=24)
    }
    else if (mElements.at(n)->getElementNumberOfNodes() == 20)
    {
        fileOut << "25 "; // VTK_QUADRATIC_HEXAHEDRON (=25)
    }
    else
    {
        fileOut << "- " ; // UNKNOWN ELEMENT TYPE
    }
}
fileOut << std::endl << " </DataArray>" << std::endl << " </Cells>" << std::endl;
<< std::endl << " </Piece>" << std::endl << " </UnstructuredGrid>" << std::endl;
<< std::endl << " </VTKFile>" << std::endl;

vector<Node *> Mesh::getNodes()
{
    return mNodes;
}

vector<Element *> Mesh::getElements()
{
    return mElements;
}

multimap<int,int> Mesh::getMapMeshDof()
{
    return mMapMeshDof;
}
```

92
multimap<int,int> Mesh::mapMeshActiveDof()
{
    mMapMeshActiveDof = mMapMeshDof;
    mActiveDofs = 0;

    for (mapIt=mMapMeshActiveDof.begin(); mapIt!=mMapMeshActiveDof.end(); mapIt++)
    {
        if (mapIt->first == mSetLockedXYZ.find(mapIt->first) &&
            mSetLockedXYZ.end() != mSetLockedXYZ.find(mapIt->first))
        {
            mapIt->second = 0; mapIt++;
            mapIt->second = 0; mapIt++;
            mapIt->second = 0;
        }
        else
        {
            mActiveDofs++;
            mapIt->second = mActiveDofs;
        }
    }
    return mMapMeshActiveDof;
}

int Mesh::getTotActiveDofs()
{
    return mActiveDofs;
}

multimap<int,int> Mesh::getMapElSurfaceLoad()
{
    return mMapElSurfaceLoad;
}

multimap<int,int> Mesh::getMapElSurfaceSide()
{
    return mMapElSurfaceDomainEdge;
}

void Mesh::newFaceFourNodes(int N1,int N2,int N3,int N4)
{
    mFaceNodes.push_back(mElementTop.at(N1));
    mFaceNodes.push_back(mElementTop.at(N2));
    mFaceNodes.push_back(mElementTop.at(N3));
    mFaceNodes.push_back(mElementTop.at(N4));
    mFaces.push_back(new Face(mFaceNumber,mFaceNodes));
    mFaceNodes.clear();
    mElementFaces.push_back(mFaces.at(mFaceNumber));
    mFaceNumber++;
}

void Mesh::newFaceThreeNodes(int N1,int N2,int N3)
{
    mFaceNodes.push_back(mElementTop.at(N1));
    mFaceNodes.push_back(mElementTop.at(N2));
    mFaceNodes.push_back(mElementTop.at(N3));
    mFaces.push_back(new Face(mFaceNumber,mFaceNodes));
mFaceNodes.clear();
mElementFaces.push_back(mFaces.at(mFaceNumber));
mFaceNumber++;
}

Source Code 6: load.h

#ifndef LOAD_H
#define LOAD_H

#include <map>
#include "mesh.h"
#include "element.h"

using std::multimap;
using std::set;
using std::string;

class Load
{
 public:
  Load(){};
  ~Load(){};
  virtual multimap<int,double> getElNodalLoading(Element *);
  virtual multimap<int,double> getMeshElNodalLoading(Mesh *);
};

class BodyForce : public Load
{
 public:
  BodyForce(double,double,double);
  ~BodyForce();
  multimap<int,double> getElNodalLoading(Element *);
 private:
  vector<double> mvBodyForce;
};

class SurfaceTraction : public Load
{
 public:
  SurfaceTraction(multimap<int,int>*,double,double,double,double,double,double,double);
  ~SurfaceTraction();
  multimap<int,double> getMeshElNodalLoading(Mesh *);
 private:
  vector<double> mvStressState;
  multimap<int,int> mMap;
};

class BoundaryDamping : public Load
{
 public:
  BoundaryDamping(multimap<int,int>*,double,double,double,double,double,double,double);
  ~BoundaryDamping();
  multimap<int,double> getMeshElNodalLoading(Mesh *);
 private:
vector<double> mvDamping;
multimap<int,int> mMap;
};
#endif

Source Code 7: load.cpp

#include "load.h"

multimap<int,double> Load::getElNodalLoading(Element *){}
multimap<int,double> Load::getMeshElNodalLoading(Mesh *){}

BodyForce::BodyForce(double accX,double accY,double accZ)
{
    mvBodyForce.push_back(accX);
    mvBodyForce.push_back(accY);
    mvBodyForce.push_back(accZ);
}

BodyForce::~BodyForce()
{
    mvBodyForce.clear();
}

multimap<int,double> BodyForce::getElNodalLoading(Element *pElement)
{
    vector<double> *mpvLoad;
    multimap<int,double> mMapLoad;
    multimap<int,double>::iterator mapItLoad;
    
    mpvLoad = pElement->getElementEqvNodalLoadBodyForce(&mvBodyForce);
    
    int nNode;
    for (int n=0; n<pElement->getElementNumberOfNodes(); n++)
    {
        nNode = pElement->getElementNodes(n+1);
        if (mMapLoad.count(nNode) == 0)
        {
            mMapLoad.insert(pair<int,double>(nNode,mpvLoad->at(3*n)));
            mMapLoad.insert(pair<int,double>(nNode,mpvLoad->at(3*n+1)));
            mMapLoad.insert(pair<int,double>(nNode,mpvLoad->at(3*n+2)));
        }
        else
        {
            mapItLoad = mMapLoad.find(nNode);
            mapItLoad->second = mapItLoad->second + mpvLoad->at(3*n);
            mapItLoad++;
            mapItLoad->second = mapItLoad->second + mpvLoad->at(3*n+1);
            mapItLoad++;
            mapItLoad->second = mapItLoad->second + mpvLoad->at(3*n+2);
        }
    }
}
delete mpvLoad;
return mMapLoad;
}

SurfaceTraction::SurfaceTraction(multimap<int,int>∗ map,double sigmaX,double sigmaY,double sigmaZ,double tauXY,double tauZX,double tauYZ)
{
    mMap = ∗map;
    mvStressState.push_back(sigmaX);
    mvStressState.push_back(sigmaY);
    mvStressState.push_back(sigmaZ);
    mvStressState.push_back(tauXY);
    mvStressState.push_back(tauZX);
    mvStressState.push_back(tauYZ);
}

SurfaceTraction::~SurfaceTraction()
{
    mvStressState.clear();
    mMap.clear();
}

multimap<int,double> SurfaceTraction::getMeshElNodalLoading(Mesh ∗pMesh)
{
    vector<double> ∗mpvLoad;
    multimap<int,double> mMapLoad;
    multimap<int,double>::iterator mapItLoad;
    multimap<int,int>::iterator mapIt;
    int nNode;
    for (mapIt=mMap.begin(); mapIt!=mMap.end(); mapIt++)
    {
        mpvLoad = pMesh->getElements().at(mapIt->first-1)->getElementEqvNodalLoadSurfaceTraction(mapIt->second,mvStressState);
        for (int n=0; n<pMesh->getElements().at(mapIt->first-1)->getElementFaces().at(mapIt->second-1)->getFaceNumberOfNodes(); n++)
        {
            nNode = pMesh->getElements().at(mapIt->first-1)->getElementFaces().at(mapIt->second-1)->getFaceNodeNumber(n+1);
            if (mMapLoad.count(nNode) == 0)
            {
                mMapLoad.insert(pair<int,double>(nNode,mpvLoad->at(3*n)));
                mMapLoad.insert(pair<int,double>(nNode,mpvLoad->at(3*n+1)));
                mMapLoad.insert(pair<int,double>(nNode,mpvLoad->at(3*n+2)));
            }
            else
            {
                mapItLoad = mMapLoad.find(nNode);
                mapItLoad->second = mapItLoad->second + mpvLoad->at(3*n); mapItLoad++;
                mapItLoad->second = mapItLoad->second + mpvLoad->at(3*n+1); mapItLoad++;
                mapItLoad->second = mapItLoad->second + mpvLoad->at(3*n+2);
            }
        }
    }
    delete mpvLoad;
BoundaryDamping::BoundaryDamping(multimap<int,double>∗ map, double xNormal, double yNormal, double xShear, double yShear, double zShear)
{
    mMap = ∗map;
    mvDamping.push_back(xNormal);
    mvDamping.push_back(yNormal);
    mvDamping.push_back(xShear);
    mvDamping.push_back(yShear);
    mvDamping.push_back(zShear);
}

BoundaryDamping::~BoundaryDamping()
{
    mvDamping.clear();
    mMap.clear();
}

multimap<int,double> BoundaryDamping::getMeshElNodalLoading(Mesh ∗ pMesh)
{
    vector<double>∗ mpvLoad;
    multimap<int,double> mMapLoad;
    multimap<int,double>::iterator mapItLoad;
    multimap<int,int>::iterator mapIt;
    int nNode;

    for (mapIt=mMap.begin(); mapIt!=mMap.end(); mapIt++)
    {
        mpvLoad = pMesh → getElements().at(mapIt → first →1) →
            getElementEqvNodalLoadBoundaryDamping(mapIt → second,mvDamping);

        for (int n=0; n<pMesh → getElements().at(mapIt → first →1) →
            getElementFaces().at(mapIt → second→1) →
            getFaceNumberOfNodes(); n++)
        {
            nNode = pMesh → getElements().at(mapIt → first→1) →
                getElementFaces().at(mapIt →
                    second→1) →
                getFaceNodeNumber(n+1);
            if (mMapLoad.count(nNode) == 0)
            {
                mMapLoad.insert(pair<int,double>(nNode,mpvLoad → at(3*n)));
                mMapLoad.insert(pair<int,double>(nNode,mpvLoad → at(3*n+1)));
                mMapLoad.insert(pair<int,double>(nNode,mpvLoad → at(3*n+2)));
            }
            else
            {
                mapItLoad = mMapLoad.find(nNode);
                mapItLoad → second = mapItLoad → second + mpvLoad → at(3*n); mapItLoad++;
                mapItLoad → second = mapItLoad → second + mpvLoad → at(3*n+1); mapItLoad++;
                mapItLoad → second = mapItLoad → second + mpvLoad → at(3*n+2);
            }
        }
    delete mpvLoad;
}
return mMapLoad;
}

Source Code 8: material.h

#ifndef MATERIAL_H
#define MATERIAL_H

class Material
{
    public:
        Material(){};
        ~Material(){};

        float getYoungs();
        float getPoisson();
        float getDensity();
        float getWaveSpeedPrimary();
        float getWaveSpeedSecondary();

    protected:
        float mYoungs;
        float mPoisson;
        float mDensity;
        float mWaveSpeedPrimary;
        float mWaveSpeedSecondary;

};

class Steel : public Material
{
    public:
        Steel();
};

class Concrete : public Material
{
    public:
        Concrete();
};

class Soil : public Material
{
    public:
        Soil();
};
#endif

Source Code 9: material.cpp

#include "material.h"
#include <math.h>
float Material::getYoungs()
{
    return mYoungs;
}

float Material::getPoisson()
{
    return mPoisson;
}

float Material::getDensity()
{
    return mDensity;
}

float Material::getWaveSpeedPrimary()
{
    return mWaveSpeedPrimary;
}

float Material::getWaveSpeedSecondary()
{
    return mWaveSpeedSecondary;
}

Steel::Steel()
{
    mYoungs = 210000000000;
    mPoisson = 0.30;
    mDensity = 7800;
}

Concrete::Concrete()
{
    mYoungs = 20000000000;
    mPoisson = 0.25; // //
    mDensity = 2500;
}

Soil::Soil()
{
    mYoungs = 50000000;
    mPoisson = 0.25;
    mDensity = 2000;
    mWaveSpeedPrimary = sqrt( (1.0-mPoisson)*mYoungs/( (1.0-2.0*mPoisson)*(1.0+mPoisson)*mDensity ) );
    mWaveSpeedSecondary = sqrt( mYoungs / ( 2.0*(1.0+mPoisson)*mDensity ) );
}

Source Code 10: element.h

#ifndef ELEMENT_H
#define ELEMENT_H

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

#include "face.h"
#include "node.h"
#include "material.h"
#include "matrix.h"

using std::vector;

class Element
{
    public:
        Element();
        ~Element();
        int getElementNumber();
        int getElementNodes(int);
        vector<Node *> getElementTopology();
        vector<Face *> getElementFaces();
        int getElementNumberOfNodes();
        Material * getElementMaterial();
        void setElementMaterial(Material *);

        virtual Matrix * getElementK();
        virtual Matrix * getElementM();
        virtual vector<double> getElementStressStrain(vector<double> *);
        virtual vector<double> * getElementEqvNodalLoadBodyForce(vector<double> *);
        virtual vector<double> * getElementEqvNodalLoadSurfaceTraction(int, vector<double> *);
        virtual vector<double> * getElementEqvNodalLoadBoundaryDamping(int, vector<double> *);

    protected:
        int mElementNumber;
        vector<Node *> mElementTop;
        vector<Face *> mElementFaces;
        Material * mpMaterial;
        vector<double> mStressStrain;
};

class C3D20 : public Element
{
    public:
        C3D20(int, vector<Node *>, vector<Face *>, Material *);
        ~C3D20();
};

class C3D10 : public Element
{
    public:
        C3D10(int, vector<Node *>, vector<Face *>, Material *);
        ~C3D10();
};
Source Code 11: element.cpp

#include "element.h"

Element::Element(){
    Element::~Element();
}

int Element::getElementNumber()
{
    return mElementNumber;
}

int Element::getElementNodes(int N)
{
    return mElementTop.at(N−1)->getNodeNumber();
}

vector<Node *> Element::getElementTopology()
{
    return mElementTop;
}

vector<Face *> Element::getElementFaces()
{
    return mElementFaces;
}

int Element::getElementNumberOfNodes()
{
    return mElementTop.size();
}

Material * Element::getElementMaterial()
{
    return mpMaterial;
}

void Element::setElementMaterial(Material * pMaterialNew)
{
    mpMaterial = pMaterialNew;
}

Matrix * Element::getElementK()
{
    return 0;
}

Matrix * Element::getElementM()
{
    return 0;
}
vector<double> Element::getElementStressStrain(vector<double>∗ pNodalDisp){}
vector<double>∗ Element::getElementEqvNodalLoadBodyForce(vector<double>∗ pBodyForce){}
vector<double>∗ Element::getElementEqvNodalLoadSurfaceTraction(int face,vector<double> pStressState){}
vector<double>∗ Element::getElementEqvNodalLoadBoundaryDamping(int face,vector<double> pDamping){}

C3D20::C3D20(int elementNumber,
vector<Node ∗> elementTop,
vector<Face ∗> elementFaces,
Material ∗ pMaterial )
{
    mElementNumber = elementNumber;
mElementTop = elementTop;
mElementFaces = elementFaces;
mpMaterial = pMaterial;
}

C3D20::~C3D20()
{
    mElementTop.clear();
mElementFaces.clear();
delete mpMaterial;
}

C3D10::C3D10( int elementNumber,
vector<Node ∗> elementTop,
vector<Face ∗> elementFaces,
Material ∗ pMaterial )
{
    mElementNumber = elementNumber;
mElementTop = elementTop;
mElementFaces = elementFaces;
mpMaterial = pMaterial;
}

C3D10::~C3D10()
{
    mElementTop.clear();
mElementFaces.clear();
delete mpMaterial;
}

---

Source Code 12: node.h

#ifndef NODE_H
#define NODE_H
#include <vector>
using std::vector;

class Node
{

"}
public:
    Node();
    ~Node();

    Node( int, double, double, double );

    int getNodeNumber();
    double getNodeX();
    double getNodeY();
    double getNodeZ();

private:
    int mNodeNumber;
    double mCoordinateX;
    double mCoordinateY;
    double mCoordinateZ;
};

#ifndef FACE_H
#define FACE_H

#include "node.h"

Node::Node( int nodeNumber, double coordinateX, double coordinateY, double coordinateZ )
: mNodeNumber( nodeNumber ), mCoordinateX( coordinateX ), mCoordinateY( coordinateY ),
  mCoordinateZ( coordinateZ ){}

int Node::getNodeNumber()
{
    return mNodeNumber;
}

double Node::getNodeX()
{
    return mCoordinateX;
}

double Node::getNodeY()
{
    return mCoordinateY;
}

double Node::getNodeZ()
{
    return mCoordinateZ;
}

#endif
#include <vector>
#include "node.h"

using std::vector;

class Face
{
public:
    Face();
    ~Face();
    Face(int, vector<Node *>);
    int getFaceNumber();
    vector<Node *> getFaceNodes();
    int getFaceNodeNumber(int);
    int getFaceNumberOfNodes();

private:
    int mFaceNumber;
    vector<Node *> mvFaceNodes;
};

Source Code 15: face.cpp

#include "face.h"

Face::~Face()
{
    mvFaceNodes.clear();
}

Face::Face(int faceNumber, vector<Node *> vFaceNodes)
{
    mFaceNumber = faceNumber;
    mvFaceNodes = vFaceNodes;
}

int Face::getFaceNumber()
{
    return mFaceNumber;
}

vector<Node *> Face::getFaceNodes()
{
    return mvFaceNodes;
}

int Face::getFaceNodeNumber(int N)
{
    return mvFaceNodes.at(N-1)->getNodeNumber();
}
```cpp
int Face::getFaceNumberOfNodes()
{
    return mvFaceNodes.size();
}
```

---

**Source Code 16: C3D4.h**

```cpp
#ifndef C3D4_H
#define C3D4_H

#include <iostream>
#include <vector>
#include <math.h>
#include "element.h"

using std::vector;

class C3D4 : public Element
{
    public:
        C3D4(int, vector<Node *>, vector<Face *>, Material *);
        ~C3D4();
        Matrix * getElementK();
        Matrix * getElementM();
        vector<double> getElementStressStrain(vector<double> *);
        vector<double> * getElementEqvNodalLoadBodyForce(vector<double> *);
        vector<double> * getElementEqvNodalLoadSurfaceTraction(int, vector<double>);
        vector<double> * getElementEqvNodalLoadBoundaryDamping(int, vector<double>);

    private:
        void fillStrainDispMat(Matrix *, double, vector<double> *, vector<double> *, vector<double> *);
        void fillConstitutiveMat(Matrix *);
        double shape1(double, double, double);
        double shape2(double, double, double);
        double shape3(double, double, double);
        double shape4(double, double, double);
};

#endif
```

---

**Source Code 17: C3D4.cpp**

```cpp
#include "C3D4.h"

C3D4::C3D4(int elementNumber, vector<Node *> elementTop, vector<Face *> elementFaces, Material * pMaterial)
{
    mElementNumber = elementNumber;
    mElementTop = elementTop;
    mElementFaces = elementFaces;
    mpMaterial = pMaterial;
}
```
APPENDIX B. SOURCE CODE

C3D4::~C3D4()
{
    mStressStrain.clear();
    mElementTop.clear();
    mElementFaces.clear();
    delete mpMaterial;
}

Matrix * C3D4::getElementK()
{
    Matrix * mpE = new Matrix(6,6);
    Matrix * mpB = new Matrix(6,12);
    Matrix * mpJ = new Matrix(3,3);
    Matrix * mpVolume = new Matrix(1,1);
    Matrix * mpKel;
    Matrix * mpKelA;
    Matrix * mpKelB;
    double detJ;
    //********************************************************************************
    vector<double> * x = new vector<double>;
    vector<double> * y = new vector<double>;
    vector<double> * z = new vector<double>;
    for(int n=0;n<getElementNumberOfNodes();n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
        y->push_back(mElementTop.at(n)->getNodeY());
        z->push_back(mElementTop.at(n)->getNodeZ());
    }
    for(int n=1;n<getElementNumberOfNodes();n++)
    {
        mpJ->setMatValue(n-1,0,(x->at(n)-x->at(0)));
        mpJ->setMatValue(n-1,1,(y->at(n)-y->at(0)));
        mpJ->setMatValue(n-1,2,(z->at(n)-z->at(0)));
    }
    detJ = mpJ->detMat();
    if(detJ <= 0)
    { std::cout << "C3D4 is coplanar! (detJ<=0)" << std::endl; }
    fillConstitutiveMat(mpE);
    fillStrainDispMat(mpB,detJ,x,y,z);
    Matrix * mpBT = mpB->transposeMat();
    mpVolume->setMatValue(0,0,detJ/6);
    mpKelA = mpE->multMat(mpB);
    mpKelB = mpBT->multMat(mpKelA);
    mpKel = mpKelB->multMat(mpVolume);
    //********************************************************************************
    delete x;
    delete y;
    delete z;
    delete mpVolume;
    delete mpJ;
    delete mpE;
    delete mpB;
    delete mpBT;
    delete mpKelA;
}
Matrix * C3D4::getElementM()
{
    Matrix * mpN = new Matrix(3,12);
    Matrix * mpNT;
    Matrix * mpMass = new Matrix(1,1);
    Matrix * mpJ = new Matrix(3,3);
    Matrix * mpMel;
    Matrix * mpMelA;
    float Rho;
    double detJ;
    //**************************************************************************
    for (int n=0; n<3; n++)
    {
        mpN->setMatValue(n,0+n,0.25);
        mpN->setMatValue(n,3+n,0.25);
        mpN->setMatValue(n,6+n,0.25);
        mpN->setMatValue(n,9+n,0.25);
    }
    //**************************************************************************
    vector<double> * x = new vector<double>;
    vector<double> * y = new vector<double>;
    vector<double> * z = new vector<double>;
    for(int n=0;n<getElementNumberOfNodes();n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
        y->push_back(mElementTop.at(n)->getNodeY());
        z->push_back(mElementTop.at(n)->getNodeZ());
    }
    //**************************************************************************
    for(int n=1;n<getElementNumberOfNodes();n++)
    {
        mpJ->setMatValue(n-1,0,(x->at(n)-x->at(0)));
        mpJ->setMatValue(n-1,1,(y->at(n)-y->at(0)));
        mpJ->setMatValue(n-1,2,(z->at(n)-z->at(0)));
    }
    detJ = mpJ->detMat();
    if(detJ <= 0)
    { std::cout << "C3D4 is coplanar! (detJ<=0)" << std::endl; }
    //**************************************************************************
    Rho = mpMaterial->getDensity();
    mpMass->setMatValue(0,0, detJ*Rho/6);
    mpNT = mpN->transposeMat();
    mpMelA = mpNT->multMat(mpN);
    mpMel = mpMelA->multMat(mpMass);
    //**************************************************************************
    delete mpMass;
    delete mpN;
    delete mpNT;
    delete mpJ;
    delete mpMelA;
    return mpMel;
}
vector<double> C3D4::getElementStressStrain(vector<double>* mNodeDisp)
{
    Matrix* mpE = new Matrix(6,6);
    Matrix* mpB = new Matrix(6,12);
    Matrix* mpJ = new Matrix(3,3);
    Matrix* mpStrain;
    Matrix* mpStress;
    Matrix* mpNodeDisp = new Matrix(12,1);
    vector<double> mStressStrain(12,0.0);
    vector<double>* x = new vector<double>;
    vector<double>* y = new vector<double>;
    vector<double>* z = new vector<double>;
    double detJ;
    //******************************************************************************
    for(int n=0;n<4;n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
        y->push_back(mElementTop.at(n)->getNodeY());
        z->push_back(mElementTop.at(n)->getNodeZ());
    }
    //******************************************************************************
    for(int n=1;n<getElementNumberOfNodes();n++)
    {
        mpJ->setMatValue(n-1,0,(x->at(n)-x->at(0)));
        mpJ->setMatValue(n-1,1,(y->at(n)-y->at(0)));
        mpJ->setMatValue(n-1,2,(z->at(n)-z->at(0)));
    }
detJ = mpJ->detMat();
    if(detJ <= 0)
    {
        std::cout << "C3D4 is coplanar! (detJ<=0)" << std::endl;
    }
    //******************************************************************************
    fillConstitutiveMat(mpE);
    fillStrainDispMat(mpB,detJ,x,y,z);
    for(unsigned int n=0;n<3*mElementTop.size();n++)
    {
        mpNodeDisp->setMatValue(n,0,mNodeDisp->at(n));
    }
    mpStrain = mpB->multMat(mpNodeDisp);
    mpStress = mpE->multMat(mpStrain);
    for(int n=0;n<6;n++)
    {
        mStressStrain.at(n) = mpStrain->getMatValue(n,0);
        mStressStrain.at(6+n) = mpStress->getMatValue(n,0);
    }
    //******************************************************************************
    delete x;
    delete y;
    delete z;
    delete mpStress;
    delete mpStrain;
    delete mpNodeDisp;
    delete mpE;
    delete mpB;
    delete mpJ;
    return mStressStrain;
}
vector<double>* C3D4::getElementEqvNodalLoadBodyForce(vector<double>* pBodyForce) {
    vector<double>* eqvNodalLoad = new vector<double>(12, 0.0);
    Matrix* mpBF = new Matrix(3, 1);
    Matrix* mpN = new Matrix(12, 3);
    Matrix* mpJ = new Matrix(3, 3);
    Matrix* mpV = new Matrix(1, 1);
    Matrix* mpR;
    Matrix* mpRA;
    vector<double>* x = new vector<double>;
    vector<double>* y = new vector<double>;
    vector<double>* z = new vector<double>;
    double detJ;
    //***************************************************************************
    for (int n=0; n<4; n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
        y->push_back(mElementTop.at(n)->getNodeY());
        z->push_back(mElementTop.at(n)->getNodeZ());
    }
    //***************************************************************************
    for (int n=0; n<3; n++)
    {
        mpN->setMatValue(0+n,n,1);
        mpN->setMatValue(3+n,n,1);
        mpN->setMatValue(6+n,n,1);
        mpN->setMatValue(9+n,n,1);
    }
    //***************************************************************************
    for(int n=1;n<getElementNumberOfNodes();n++)
    {
        mpJ->setMatValue(n-1,0,(x->at(n)-x->at(0)));
        mpJ->setMatValue(n-1,1,(y->at(n)-y->at(0)));
        mpJ->setMatValue(n-1,2,(z->at(n)-z->at(0)));
    }
    detJ = mpJ->detMat();
    if(detJ <= 0){ std::cout << "C3D4 is coplanar! (detJ<=0)" << std::endl; }
    //***************************************************************************
    mpV->setMatValue(0,0,0.25*mpMaterial->getDensity()*detJ/6);
    for (int n=0; n<3; n++)
    {
        mpBF->setMatValue(n,0,pBodyForce->at(n));
    }
    mpRA = mpBF->multMat(mpV);
    mpR = mpN->multMat(mpRA);
    for (int n=0; n<12; n++)
    {
        eqvNodalLoad->at(n) = mpR->getMatValue(n,0);
    }
    //***************************************************************************
    delete x;
    delete y;
    delete z;
    delete mpBF;
    delete mpN;
    delete mpV;
    delete mpRA;
}
APPENDIX B. SOURCE CODE

```cpp
delete mpR;
delete mpJ;
return eqvNodalLoad;
}

vector<double> C3D4::getElementEqvNodalLoadSurfaceTraction(int face,vector<double> pStressState)
{
    vector<double> eqvNodalLoad = new vector<double>(9,0.0);
    Matrix O = new Matrix(3,3);
    Matrix N = new Matrix(9,3);
    Matrix dA = new Matrix(3,1);
    Matrix Re;
    Matrix ReA;
    vector<double> x = new vector<double>;
    vector<double> y = new vector<double>;
    vector<double> z = new vector<double>;
    //********************************************************************************
    for(int n=0;n<4;n++)
    {
        x.push_back(mElementTop.at(n)->getNodeX());
        y.push_back(mElementTop.at(n)->getNodeY());
        z.push_back(mElementTop.at(n)->getNodeZ());
    }
    double gauss = 0.333333333;
    if (face == 1)
    {
        dA.setMatValue(0,0,((y.at(0)-y.at(2))*(z.at(0)-z.at(1))-(y.at(0)-y.at(1))*(z.at(0)-z.at(2)))/2);
        dA.setMatValue(1,0,((x.at(0)-x.at(1))*(z.at(0)-z.at(2))-(x.at(0)-x.at(1))*(z.at(0)-z.at(1)))/2);
        dA.setMatValue(2,0,((x.at(0)-x.at(1))*(y.at(0)-y.at(3))-(x.at(0)-x.at(3))*(y.at(0)-y.at(1)))/2);
        for (int n=0; n<3; n++)
        {
            N.setMatValue(n,n,shape1(gauss,gauss,0));
            N.setMatValue(3+n,n,shape2(gauss,gauss,0));
            N.setMatValue(6+n,n,shape3(gauss,gauss,0));
        }
    }
    else if (face == 2)
    {
        dA.setMatValue(0,0,((y.at(0)-y.at(2))*(z.at(0)-z.at(1))-(y.at(0)-y.at(1))*(z.at(0)-z.at(2)))/2);
        dA.setMatValue(1,0,((x.at(0)-x.at(1))*(z.at(0)-z.at(2))-(x.at(0)-x.at(1))*(z.at(0)-z.at(1)))/2);
        dA.setMatValue(2,0,((x.at(0)-x.at(1))*(y.at(0)-y.at(3))-(x.at(0)-x.at(3))*(y.at(0)-y.at(1)))/2);
        for (int n=0; n<3; n++)
        {
            N.setMatValue(n,n,shape1(gauss,0,gauss));
            N.setMatValue(3+n,n,shape2(gauss,0,gauss));
            N.setMatValue(6+n,n,shape3(gauss,0,gauss));
        }
    }
    else if (face == 3)
    
    return eqvNodalLoad;
```

110

```cpp
}{
  dA->setMatValue(0,0,((y->at(1)-y->at(2))*(z->at(1)-z->at(3))-((y->at(1)-y->at(3))*(z->at(1)-z->at(2))))/2);
  dA->setMatValue(1,0,((x->at(1)-x->at(2))*(z->at(1)-z->at(3))-((x->at(1)-x->at(3))*(z->at(1)-z->at(2))))/2);
  dA->setMatValue(2,0,((x->at(1)-x->at(3))*(y->at(1)-y->at(2))-((x->at(1)-x->at(2))*(y->at(1)-y->at(3))))/2);
  for (int n=0; n<3; n++)
  {
    N->setMatValue(n,n,shape2(gauss,gauss,gauss));
    N->setMatValue(3+n,n,shape3(gauss,gauss,gauss));
    N->setMatValue(6+n,n,shape4(gauss,gauss,gauss));
  }
}
else if (face == 4)
{
  dA->setMatValue(0,0,((y->at(0)-y->at(3))*(z->at(0)-z->at(2))-((y->at(0)-y->at(2))*(z->at(0)-z->at(3))))/2);
  dA->setMatValue(1,0,((x->at(0)-x->at(2))*(z->at(0)-z->at(3))-((x->at(0)-x->at(3))*(z->at(0)-z->at(2))))/2);
  dA->setMatValue(2,0,((x->at(0)-x->at(3))*(y->at(0)-y->at(2))-((x->at(0)-x->at(2))*(y->at(0)-y->at(3))))/2);
  for (int n=0; n<3; n++)
  {
    N->setMatValue(n,n,shape3(0,gauss,gauss));
    N->setMatValue(3+n,n,shape1(0,gauss,gauss));
    N->setMatValue(6+n,n,shape4(0,gauss,gauss));
  }
} 
else
{ 
  std::cout << "Unknown face! (C3D4::getElementEqvNodalLoad) " << std::endl;
}
//********************************************************************************
for (int n=0; n<3; n++){
  O->setMatValue(n,n,pStressState.at(n));
}
O->setMatValue(1,0,pStressState.at(3));
O->setMatValue(2,0,pStressState.at(4));
O->setMatValue(0,1,pStressState.at(3));
O->setMatValue(2,1,pStressState.at(5));
O->setMatValue(0,2,pStressState.at(4));
O->setMatValue(1,2,pStressState.at(5));
ReA = O->multMat(dA);
Re = N->multMat(ReA);
for (int n=0; n<9; n++){
  eqvNodalLoad->at(n) = Re->getMatValue(n,0);
}
//********************************************************************************
delete x;
delete y;
delete z;
delete dA;
delete O;
delete N;
delete ReA;
delete Re;
return eqvNodalLoad;
```
vector<double>* C3D4::getElementEqvNodalLoadBoundaryDamping(int face, vector<double> pDamping)
{
    vector<double>* eqvNodalLoad = new vector<double>(9, 0.0);
    Matrix* O = new Matrix(3, 3);
    Matrix* Oshear = new Matrix(3, 1);
    Matrix* N = new Matrix(9, 3);
    Matrix* dA = new Matrix(3, 1);
    Matrix* Re;
    Matrix* ReA;
    Matrix* Rshear;
    double faceArea;
    vector<double>* x = new vector<double>;
    vector<double>* y = new vector<double>;
    vector<double>* z = new vector<double>;
    double normal1, normal2, normal3, normal4, normal5;
    //**************************************************************************
    for(int n=0; n<4; n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
        y->push_back(mElementTop.at(n)->getNodeY());
        z->push_back(mElementTop.at(n)->getNodeZ());
    }
    double gauss = 0.333333333;
    //**************************************************************************
    if (face == 1)
    {
        normal1 = (y->at(0) - y->at(2)) * (z->at(0) - z->at(1)) - ((y->at(0) - y->at(1)) * (z->at(0) - z->at(2)));
        normal2 = (x->at(0) - x->at(1)) * (z->at(0) - z->at(2)) - ((x->at(0) - x->at(2)) * (z->at(0) - z->at(1)));
        normal3 = (x->at(0) - x->at(2)) * (y->at(0) - y->at(1)) - ((x->at(0) - x->at(1)) * (y->at(0) - y->at(2)));
        dA->setMatValue(0, 0, (normal1) / 2);
        dA->setMatValue(1, 0, (normal2) / 2);
        dA->setMatValue(2, 0, (normal3) / 2);
        for (int n=0; n<3; n++)
        {
            N->setMatValue(n, n, shape1(gauss, gauss, 0));
            N->setMatValue(3+n, n, shape2(gauss, gauss, 0));
            N->setMatValue(6+n, n, shape3(gauss, gauss, 0));
        }
    }
    else if (face == 2)
    {
        normal1 = (y->at(0) - y->at(1)) * (z->at(0) - z->at(3)) - ((y->at(0) - y->at(3)) * (z->at(0) - z->at(1)));
        normal2 = (x->at(0) - x->at(3)) * (z->at(0) - z->at(1)) - ((x->at(0) - x->at(1)) * (z->at(0) - z->at(3)));
        normal3 = (x->at(0) - x->at(1)) * (y->at(0) - y->at(3)) - ((x->at(0) - x->at(3)) * (y->at(0) - y->at(1)));
        dA->setMatValue(0, 0, (normal1) / 2);
        dA->setMatValue(1, 0, (normal2) / 2);
        dA->setMatValue(2, 0, (normal3) / 2);
    }
}
for (int n=0; n<3; n++)
{
    N->setMatValue(n,n,shape1(gauss,0,gauss));
    N->setMatValue(3+n,n,shape2(gauss,0,gauss));
    N->setMatValue(6+n,n,shape4(gauss,0,gauss));
}
else if (face == 3)
{
    normal1 = (y->at(1)-y->at(2))*(z->at(1)-z->at(3)) - (y->at(1)-y->at(3))*z->at(2);
    normal2 = (x->at(1)-x->at(3))*(z->at(1)-z->at(2)) - (x->at(1)-x->at(2))*z->at(3);
    normal3 = (x->at(1)-x->at(2))*(y->at(1)-y->at(3)) - (x->at(1)-x->at(3))*y->at(2);
    dA->setMatValue(0,0,(normal1)/2);
    dA->setMatValue(1,0,(normal2)/2);
    dA->setMatValue(2,0,(normal3)/2);
    for (int n=0; n<3; n++)
    {
        N->setMatValue(n,n,shape2(gauss,gauss,gauss));
        N->setMatValue(3+n,n,shape3(gauss,gauss,gauss));
        N->setMatValue(6+n,n,shape4(gauss,gauss,gauss));
    }
}
else if (face == 4)
{
    normal1 = (y->at(0)-y->at(3))*(z->at(0)-z->at(2)) - (y->at(0)-y->at(2))*z->at(3);
    normal2 = (x->at(0)-x->at(2))*(z->at(0)-z->at(3)) - (x->at(0)-x->at(3))*z->at(2);
    normal3 = (x->at(0)-x->at(2))*(y->at(0)-y->at(3)) - (x->at(0)-x->at(3))*y->at(2);
    dA->setMatValue(0,0,(normal1)/2);
    dA->setMatValue(1,0,(normal2)/2);
    dA->setMatValue(2,0,(normal3)/2);
    for (int n=0; n<3; n++)
    {
        N->setMatValue(n,n,shape3(0,gauss,gauss,gauss));
        N->setMatValue(3+n,n,shape1(0,gauss,gauss,gauss));
        N->setMatValue(6+n,n,shape4(0,gauss,gauss,gauss));
    }
}
else
{
    std::cout << "Unknown face! (C3D4::getElementEqvNodalLoadBondaryDamping) " << std::endl;
}

//*******************************
faceArea = dA->detMat();
O->setMatValue(0,0,pDamping.at(0));
O->setMatValue(1,1,pDamping.at(1));
O->setMatValue(0,1,pDamping.at(2));
O->setMatValue(1,0,pDamping.at(3));
Oshear->setMatValue(2,0,faceArea*pDamping.at(4));
ReA = O->multMat(dA);
Re = N->multMat(ReA);
Rshear = N->multMat(Oshear);
for (int n=0; n<9; n++) {
    eqvNodalLoad = Re�->getMatValue(n,0) + Rshear�->getMatValue(n,0);
}

delete x;
delete y;
delete z;
delete dA;
delete N;
delete ReA;
delete Re;
delete Rshear;
return eqvNodalLoad;

void C3D4::fillStrainDispMat(Matrix * mpB, double detJ, vector<double> * x, vector<double> * y, vector<double> * z) {
    mpB�->setMatValue(0,0,((y�->at(3)�−y�->at(1))�∗(z�->at(2)�−z�->at(1))�−(y�->at(2)�−y�->at (1))�∗(z�->at(3)�−z�->at(1))))�/detJ);
    mpB�->setMatValue(1,1,((x�->at(2)�−x�->at(1))�∗(z�->at(3)�−z�->at(1))�−(x�->at(3)�−x�->at (1))�∗(z�->at(2)�−z�->at(1))))�/detJ);
    mpB�->setMatValue(2,2,((x�->at(3)�−x�->at(1))�∗(y�->at(2)�−y�->at(1))�−(x�->at(2)�−x�->at (1))�∗(y�->at(3)�−y�->at(1))))�/detJ);
    mpB�->setMatValue(3,0,mpB�->getMatValue(1,1));
    mpB�->setMatValue(5,0,mpB�->getMatValue(2,2));
    mpB�->setMatValue(3,1,mpB�->getMatValue(0,0));
    mpB�->setMatValue(4,1,mpB�->getMatValue(2,2));
    mpB�->setMatValue(4,2,mpB�->getMatValue(1,1));
    mpB�->setMatValue(5,2,mpB�->getMatValue(0,0));
    mpB�->setMatValue(0,3,((y�->at(2)�−y�->at(0))�∗(z�->at(3)�−z�->at(2))�−(y�->at(2)�−y�->at (1))�∗(z�->at(0)�−z�->at(2))))�/detJ);
    mpB�->setMatValue(1,4,((x�->at(3)�−x�->at(2))�∗(z�->at(2)�−z�->at(0))�−(x�->at(0)�−x�->at (2))�∗(z�->at(2)�−z�->at(3))))�/detJ);
    mpB�->setMatValue(2,5,((x�->at(2)�−x�->at(0))�∗(y�->at(3)�−y�->at(2))�−(x�->at(2)�−x�->at (1))�∗(y�->at(0)�−y�->at(2))))�/detJ);
    mpB�->setMatValue(3,3,mpB�->getMatValue(1,4));
    mpB�->setMatValue(5,3,mpB�->getMatValue(2,5));
    mpB�->setMatValue(3,4,mpB�->getMatValue(0,3));
    mpB�->setMatValue(4,4,mpB�->getMatValue(2,5));
    mpB�->setMatValue(4,5,mpB�->getMatValue(1,4));
    mpB�->setMatValue(5,5,mpB�->getMatValue(0,3));
    mpB�->setMatValue(0,6,((y�->at(1)�−y�->at(3))�∗(z�->at(0)�−z�->at(3))�−(y�->at(0)�−y�->at (1))�∗(z�->at(1)�−z�->at(3))))�/detJ);
    mpB�->setMatValue(1,7,((x�->at(0)�−x�->at(3))�∗(z�->at(1)�−z�->at(3))�−(x�->at(1)�−x�->at (3))�∗(z�->at(0)�−z�->at(3))))�/detJ);
    mpB�->setMatValue(2,8,((x�->at(1)�−x�->at(3))�∗(y�->at(0)�−y�->at(3))�−(x�->at(0)�−x�->at (3))�∗(y�->at(1)�−y�->at(3))))�/detJ);
    mpB�->setMatValue(3,6,mpB�->getMatValue(1,7));
    mpB�->setMatValue(5,6,mpB�->getMatValue(2,8));
    mpB�->setMatValue(3,7,mpB�->getMatValue(0,6));
    mpB�->setMatValue(4,7,mpB�->getMatValue(2,8));
    mpB�->setMatValue(4,8,mpB�->getMatValue(1,7));
}
mpB->setMatValue(5,8,mpB->getMatValue(0,6));
//????????????????????????????????????????????????????????
mpB->setMatValue(0,9,((y->at(0)−y->at(2))*(z->at(1)−z->at(0))−(y->at(0)−y->at(1))*(z->at(2)−z->at(0))))/detJ);
mpB->setMatValue(1,10,((x->at(1)−x->at(0))*(z->at(0)−z->at(2))−(x->at(0)−x->at(1))*(z->at(2)−z->at(0))))/detJ);
mpB->setMatValue(2,11,((x->at(0)−x->at(2))*(y->at(1)−y->at(0))−(x->at(0)−x->at(1))*(y->at(2)−y->at(0))))/detJ);
mpB->setMatValue(3,9,mpB->getMatValue(1,10));
mpB->setMatValue(5,9,mpB->getMatValue(2,11));
mpB->setMatValue(3,10,mpB->getMatValue(0,9));
mpB->setMatValue(4,10,mpB->getMatValue(2,11));
mpB->setMatValue(4,11,mpB->getMatValue(1,10));
mpB->setMatValue(5,11,mpB->getMatValue(0,9));

void C3D4::fillConstitutiveMat(Matrix * mpE)
{
    float v = mpMaterial->getPoisson();
    float E = mpMaterial->getYoungs();
    double c = E/((1+v)*(1−2*v));
    double G = E/(2*(1+v));
    //????????????????????????????????????????????????????????
    for(int n=0;n<3;n++){
        mpE->setMatValue(n,n,(1−v)*c);
    } for(int n=3;n<6;n++){
        mpE->setMatValue(n,n,G);
    }
    mpE->setMatValue(0,1,v*c);
    mpE->setMatValue(1,0,v*c);
    mpE->setMatValue(0,2,v*c);
    mpE->setMatValue(2,0,v*c);
    mpE->setMatValue(1,2,v*c);
    mpE->setMatValue(2,1,v*c);
}

double C3D4::shape1(double r, double s, double t)
{
    return 1−r−s−t;
}

double C3D4::shape2(double r, double s, double t)
{
    return r;
}

double C3D4::shape3(double r, double s, double t)
{
    return s;
}

double C3D4::shape4(double r, double s, double t)
{
    return t;
}
Source Code 18: C3D8.h

```cpp
#ifndef C3D8_H
#define C3D8_H

#include <iostream>
#include <vector>
#include "element.h"
using std::vector;

class C3D8 : public Element
{
public:
    C3D8(int, vector<Node *>, vector<Face *>, Material *);
    ~C3D8();

    Matrix * getElementK();
    vector<double> * getElementStressStrain(vector<double> *);
    vector<double> * getElementEqvNodalLoadSurfaceTraction(int, vector<double> *);
    vector<double> * getElementEqvNodalLoadBodyForce(vector<double> *);

private:

    Matrix * setKel(Matrix *, Matrix *, Matrix *, Matrix *, double, double, double);
    void setJacobian(Matrix *, double, double, double, vector<double> *, vector<double> *);
    void setDifShapeFunctions(Matrix *, double, double, double);
    double jac1(double, double, vector<double> *);
    double jac2(double, double, vector<double> *);
    double jac3(double, double, vector<double> *);
    double dNf1(double, double);
    double dNf2(double, double);
    double dNf3(double, double);
    double dNf4(double, double);
    void fillConstitutiveMat(Matrix *);
    void fillStrainDisp(Matrix *);
};

#endif
```

Source Code 19: C3D8.cpp

```cpp
#include "C3D8.h"

C3D8::C3D8(int elementNumber,
           vector<Node *> elementTop,
           vector<Face *> elementFaces,
           Material * pMaterial)
{
    mElementNumber = elementNumber;
    mElementTop.push_back(elementTop.at(4));
    mElementTop.push_back(elementTop.at(0));
    mElementTop.push_back(elementTop.at(3));
    mElementTop.push_back(elementTop.at(7));
    mElementTop.push_back(elementTop.at(5));
}```
mElementTop.push_back(elementTop.at(1));
mElementTop.push_back(elementTop.at(2));
mElementTop.push_back(elementTop.at(6));

mElementFaces = elementFaces;
mpMaterial = pMaterial;
}

C3D8::~C3D8()
{
    mElementTop.clear();
    mElementFaces.clear();
    delete mpMaterial;
}

Matrix * C3D8::getElementK()
{
    Matrix * mpE = new Matrix(6,6);
    Matrix * mpA = new Matrix(6,9);
    Matrix * mpJac = new Matrix(3,3);
    Matrix * mpJacInvBig = new Matrix(9,9);
    Matrix * mpJacDet = new Matrix(1,1);
    Matrix * mpKel = new Matrix(24,24);

    vector<double> * x = new vector<double>;
    vector<double> * y = new vector<double>;
    vector<double> * z = new vector<double>;

    //************
    for (int n=0; n<8; n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
    }
    for (int n=0; n<8; n++)
    {
        y->push_back(mElementTop.at(n)->getNodeY());
    }
    for (int n=0; n<8; n++)
    {
        z->push_back(mElementTop.at(n)->getNodeZ());
    }

    setJacobian(mpJac,0,0,0,x,y,z);
    mpJacInv = mpJac->invMat();
    mpJacDet->setMatValue(0,0,mpJac->detMat());
    //************
    for (int n=0; n<3; n++)
    {
        for (int m=0; m<3; m++)
        {
            mpJacInvBig->setMatValue(n*3,n*3+m,mpJacInv->getMatValue(0,m));
            mpJacInvBig->setMatValue(n*3+1,n*3+m,mpJacInv->getMatValue(1,m));
            mpJacInvBig->setMatValue(n*3+2,n*3+m,mpJacInv->getMatValue(2,m));
        }
    }

    fillConstitutiveMat(mpE);
    fillStrainDisp(mpA);

    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,0.5773503,0.5773503,0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,0.5773503,0.5773503,-0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,0.5773503,-0.5773503,-0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,0.5773503,-0.5773503,0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,-0.5773503,-0.5773503,-0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,-0.5773503,-0.5773503,0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,0.5773503,0.5773503,-0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,0.5773503,0.5773503,0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,-0.5773503,0.5773503,-0.5773503));
    mpKel->addMat(setKel(mpE,mpA,mpJacInvBig,mpJacDet,-0.5773503,0.5773503,0.5773503));
}
APPENDIX B. SOURCE CODE

```cpp
// delete x;
delete y;
delete z;
delete mpA;
delete mpE;
delete mpJacInvBig;
delete mpJacInv;
delete mpJacDet;
delete mpJac;
return mpKel;
}

Matrix * C3D8::setKel( Matrix * mpE, Matrix * mpA, Matrix * mpJacInvExp, Matrix * mpJacDet,
    double Xi, double Eta, double Zeta)
{
    Matrix * mpN = new Matrix(9,24);
    Matrix * mpB = new Matrix(6,24);
    Matrix * mpBT = new Matrix(24,6);
    Matrix * mpKelGP = new Matrix(24,24);
    //**************
    setDifShapeFunctions(mpN,Xi,Eta,Zeta);
    //**************
    mpB = mpA - multMat(mpJacInvExp - multMat(mpN));
    mpBT = mpB - transposeMat();
    mpKelGP = mpBT - multMat(mpE - multMat(mpB));
    mpKelGP = mpKelGP - multMat(mpJacDet);
    //**************
    delete mpB;
    delete mpN;
    delete mpBT;
    return mpKelGP;
}

void C3D8::setJacobian( Matrix * mpJac, double Xi, double Eta, double Zeta, vector<double> * x,
    vector<double> * y,vector<double> * z)
{
    mpJac->setMatValue(0,0,jac1(Eta,Zeta,x));
    mpJac->setMatValue(1,0,jac2(Xi,Zeta,x));
    mpJac->setMatValue(2,0,jac3(Xi,Eta,x));
    mpJac->setMatValue(0,1,jac1(Eta,Zeta,y));
    mpJac->setMatValue(1,1,jac2(Xi,Zeta,y));
    mpJac->setMatValue(2,1,jac3(Xi,Eta,y));
    mpJac->setMatValue(0,2,jac1(Eta,Zeta,z));
    mpJac->setMatValue(1,2,jac2(Xi,Zeta,z));
    mpJac->setMatValue(2,2,jac3(Xi,Eta,z));
}

void C3D8::setDifShapeFunctions( Matrix * mpN, double Xi, double Eta, double Zeta)
{
    for(int n=0;n<3;n++)
    {
        mpN->setMatValue(3*n,n,dNf3(Eta,Zeta));
        mpN->setMatValue(3*n+1,n,dNf3(Xi,Zeta));
        mpN->setMatValue(3*n+2,n,dNf4(Xi,Eta));
        mpN->setMatValue(3*n+3,-dNf4(Eta,Zeta));
    }
```
mN->setMatValue(3*n+1,n+3,-dNf4(Xi,Zeta));
mN->setMatValue(3*n+2,n+3,-dNf4(Xi,Zeta));
mN->setMatValue(3*n+3,n+6,dNf2(Eta,Zeta));
mN->setMatValue(3*n+1,n+6,dNf3(Xi,Zeta));
mN->setMatValue(3*n+2,n+6,dNf3(Xi,Eta));
mN->setMatValue(3*n,n+9,-dNf1(Eta,Zeta));
mN->setMatValue(3*n+1,n+9,-dNf3(Xi,Zeta));
mN->setMatValue(3*n+2,n+9,-dNf3(Xi,Eta));
mN->setMatValue(3*n,n+12,dNf1(Eta,Zeta));
mN->setMatValue(3*n+1,n+12,-dNf3(Eta,Zeta));
mN->setMatValue(3*n+2,n+12,-dNf2(Eta,Zeta));
mN->setMatValue(3*n,n+15,dNf2(Xi,Eta));
mN->setMatValue(3*n+1,n+15,dNf4(Xi,Eta));
mN->setMatValue(3*n+2,n+15,dNf2(Xi,Eta));
mN->setMatValue(3*n,n+18,-dNf2(Eta,Zeta));
mN->setMatValue(3*n+1,n+18,-dNf2(Xi,Zeta));
mN->setMatValue(3*n+2,n+18,-dNf1(Xi,Eta));
mN->setMatValue(3*n,n+21,dNf1(Eta,Zeta));
mN->setMatValue(3*n+1,n+21,dNf1(Xi,Zeta));
mN->setMatValue(3*n+2,n+21,dNf1(Xi,Eta));
}

double C3D8::jac1(double mValue1,double mValue2,vector<double>* mvCoordinates) {
    return (mvCoordinates->at(0)*dNf3(mValue1,mValue2) -
            mvCoordinates->at(1)*dNf4(mValue1,mValue2) +
            mvCoordinates->at(2)*dNf2(mValue1,mValue2) -
            mvCoordinates->at(3)*dNf4(mValue1,mValue2) -
            mvCoordinates->at(4)*dNf3(mValue1,mValue2) +
            mvCoordinates->at(5)*dNf4(mValue1,mValue2) -
            mvCoordinates->at(6)*dNf2(mValue1,mValue2) +
            mvCoordinates->at(7)*dNf1(mValue1,mValue2));
}

double C3D8::jac2(double mValue1,double mValue2,vector<double>* mvCoordinates) {
    return (mvCoordinates->at(0)*dNf4(mValue1,mValue2) -
            mvCoordinates->at(1)*dNf4(mValue1,mValue2) +
            mvCoordinates->at(2)*dNf4(mValue1,mValue2) -
            mvCoordinates->at(3)*dNf1(mValue1,mValue2) -
            mvCoordinates->at(4)*dNf3(mValue1,mValue2) +
            mvCoordinates->at(5)*dNf2(mValue1,mValue2) -
            mvCoordinates->at(6)*dNf2(mValue1,mValue2) +
            mvCoordinates->at(7)*dNf1(mValue1,mValue2));
}

double C3D8::jac3(double mValue1,double mValue2,vector<double>* mvCoordinates) {
    return (mvCoordinates->at(0)*dNf4(mValue1,mValue2) -
            mvCoordinates->at(1)*dNf4(mValue1,mValue2) +
            mvCoordinates->at(2)*dNf4(mValue1,mValue2) -
            mvCoordinates->at(3)*dNf3(mValue1,mValue2) -
            mvCoordinates->at(4)*dNf2(mValue1,mValue2) +
            mvCoordinates->at(5)*dNf2(mValue1,mValue2) -
            mvCoordinates->at(6)*dNf1(mValue1,mValue2) +
            mvCoordinates->at(7)*dNf1(mValue1,mValue2));
}
APPENDIX B. SOURCE CODE

mvCoordinates->at(7)*dNf1(mValue1,mValue2));
}

double C3D8::dNf1(double mValue1,double mValue2)
{
    return (mValue1+1)*(mValue2+1)/8;
}

double C3D8::dNf2(double mValue1,double mValue2)
{
    return (mValue1+1)*(mValue2-1)/8;
}

double C3D8::dNf3(double mValue1,double mValue2)
{
    return (mValue1-1)*(mValue2+1)/8;
}

double C3D8::dNf4(double mValue1,double mValue2)
{
    return (mValue1-1)*(mValue2-1)/8;
}

vector<double> C3D8::getElementStressStrain(vector<double> * mNodeDisp)
{
    Matrix * mpE = new Matrix(6,6);
    Matrix * mpA = new Matrix(6,9);
    Matrix * mpJac = new Matrix(3,3);
    Matrix * mpJacDet = new Matrix(1,1);
    Matrix * mpJacInvBig = new Matrix(9,9);
    Matrix * mpN = new Matrix(9,24);
    Matrix * mpB = new Matrix(6,24);
    Matrix * mpStrain = new Matrix(6,1);
    Matrix * mpStress = new Matrix(6,1);
    Matrix * mpNodeDisp = new Matrix(24,1);
    vector<double> mStressStrain(12,0.0);
    vector<double> * x = new vector<double>;
    vector<double> * y = new vector<double>;
    vector<double> * z = new vector<double>;
    //*********************************************************************
    for (int n=0; n<8; n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
        y->push_back(mElementTop.at(n)->getNodeY());
        z->push_back(mElementTop.at(n)->getNodeZ());
    }
    //*********************************************************************
    setJacobian(mpJac,0,0,0,x,y,z);
    Matrix * mpJacLnv = mpJac->invMat();
    mpJacDet->setMatValue(0,0,mpJac->detMat());
    //*********************************************************************
    for (int n=0; n<3; n++)
    {
        for (int m=0; m<3; m++)
        {
            mpJacInvBig->setMatValue(n*3,n*3+m,mpJacLnv->getMatValue(0,m));
        }
    }
}
mpJacInvBig->setMatValue(n*3+1,n*3+m,mpJacInv->getMatValue(1,m));
mpJacInvBig->setMatValue(n*3+2,n*3+m,mpJacInv->getMatValue(2,m));
}
}
fillConstitutiveMat(mpE);
fillStrainDisp(mpA);
setDifShapeFunctions(mpN,0,0,0);
mpB = mpA->multMat(mpJacInv->multMat(mpN));
for (int n=0; n<24; n++)
  { mpNodeDisp->setMatValue(n,0,mNodeDisp->at(n)); }
mpStrain = mpB->multMat(mpNodeDisp);
mpStress = mpE->multMat(mpStrain);
for (int n=0; n<6; n++)
  { mStressStrain.at(n) = mpStrain->getMatValue(n,0); }
for (int n=0; n<6; n++)
  { mStressStrain.at(6+n) = mpStress->getMatValue(n,0); }
//***********************************************************************
delete x;
delete y;
delete z;
delete mpStress;
delete mpStrain;
delete mpNodeDisp;
delete mpE;
delete mpA;
delete mpJac;
delete mpJacInv;
delete mpJacInvBig;
return mStressStrain;
}

vector<double>* C3D8::getElementEqvNodalLoadBodyForce(vector<double>* pBodyForce)
{
vector<double>* eqvNodalLoad = new vector<double>(24,0.0);
Matrix * mpBF = new Matrix(3,1);
Matrix * mpN = new Matrix(24,3);
Matrix * mpJac = new Matrix(3,3);
Matrix * mpJacDet = new Matrix(1,1);
Matrix * mpR = new Matrix(24,1);
vector<double> * x = new vector<double>;
vector<double> * y = new vector<double>;
vector<double> * z = new vector<double>;
//***********************************************************************
for (int n=0; n<8; n++)
  { x->push_back(mElementTop.at(n)->getNodeX());
    y->push_back(mElementTop.at(n)->getNodeY());
    z->push_back(mElementTop.at(n)->getNodeZ());
  }
//***********************************************************************
for (int n=0; n<3; n++)
  { mpN->setMatValue(0+n,n,1);
    mpN->setMatValue(3+n,n,1);
    mpN->setMatValue(6+n,n,1);
  }
//***********************************************************************
for (int n=0; n<8; n++)
  { x->push_back(mElementTop.at(n)->getNodeX());
    y->push_back(mElementTop.at(n)->getNodeY());
    z->push_back(mElementTop.at(n)->getNodeZ());
  }
//***********************************************************************
for (int n=0; n<3; n++)
  { mpN->setMatValue(0+n,n,1);
    mpN->setMatValue(3+n,n,1);
    mpN->setMatValue(6+n,n,1);
  }
//***********************************************************************
delete x;
delete y;
delete z;
delete mpStress;
delete mpStrain;
delete mpNodeDisp;
delete mpE;
delete mpA;
delete mpJac;
delete mpJacInv;
delete mpJacInvBig;
return mStressStrain;
APPENDIX B. SOURCE CODE

```cpp
mpN->setMatValue(9+n,n,1);
mpN->setMatValue(12+n,n,1);
mpN->setMatValue(15+n,n,1);
mpN->setMatValue(18+n,n,1);
mpN->setMatValue(21+n,n,1);

} //***********************************************************************/
setJacobian(mpJac,0,0,0,x,y,z);
mpJacDet->setMatValue(0,0,mpJac->detMat()*mpMaterial->getDensity());
for (int n=0; n<3; n++)
    mpBF->setMatValue(n,0,pBodyForce->at(n));
mpR = mpN->multMat(mpBF->multMat(mpJacDet));
for (int n=0; n<24; n++)
eqvNodalLoad->at(n) = mpR->getMatValue(n,0);

/********************************************************************************
delete x;
delete y;
delete z;
delete mpBF;
delete mpN;
delete mpJac;
delete mpR;
return eqvNodalLoad;
}

vector<double> C3D8::getElementEqvNodalLoadSurfaceTraction(int face,vector<double>* pStressState)
{
    vector<double> eqvNodalLoad = new vector<double>(12,0.0);
    Matrix * mpO = new Matrix(3,3);
    Matrix * mpN = new Matrix(12,3);
    Matrix * mpJ = new Matrix(3,1);
    Matrix * mpR = new Matrix(12,1);
    vector<double> * x = new vector<double>;
    vector<double> * y = new vector<double>;
    vector<double> * z = new vector<double>;

    //***********************************************************************/
    for (int n=0; n<8; n++)
    {
        x->push_back(mElementTop.at(n)->getNodeX());
        y->push_back(mElementTop.at(n)->getNodeY());
        z->push_back(mElementTop.at(n)->getNodeZ());
    }
    //***********************************************************************/
    for (int n=0; n<3; n++)
    {
        mpN->setMatValue(0+n,n,1);
        mpN->setMatValue(3+n,n,1);
        mpN->setMatValue(6+n,n,1);
        mpN->setMatValue(9+n,n,1);
    }

    if (face == 1) {
        mpJ->setMatValue(0,0,(0.0625*(y->at(0)+y->at(1)-y->at(2)-y->at(3))*z->at(0)-z->at(1)-z->at(2)+z->at(3))-(y->at(0)-y->at(1)-y->at(2)+y->at(3))*z->at(0)+z->at(1)-z->at(2)-z->at(3)));
```
mpJ->setMatValue(1,0,(0.0625*((x->at(0)-x->at(1)-x->at(2)+x->at(3)))+(z->at(0)+z-
->at(1)-z->at(2)-z->at(3))+(x->at(0)+x->at(1)-x->at(2)-x->at(3))+(z->at(0)-z->at(1)+z->at(2)+z->at(3))));

mpJ->setMatValue(2,0,(0.0625*((x->at(0)-x->at(1)-x->at(2)-x->at(3))+y->at(0)-y-
->at(1)-y->at(2)+y->at(3))-x->at(0)-x->at(1)-x->at(2)+x->at(3))*y->at(0)+y-
->at(1)-y->at(2)-y->at(3))));

else if (face == 2) {
    mpJ->setMatValue(0,0,(0.0625*((y->at(4)-y->at(5)-y->at(6)+y->at(7))+(z->at(4)+z-
->at(5)-z->at(6)-z->at(7))+(y->at(4)+y->at(5)-y->at(6)-y->at(7))+(z->at(4)-at(5)-z->at(6)+z->at(7))));

mpJ->setMatValue(1,0,(0.0625*((x->at(4)+x->at(5)-x->at(6)-x->at(7))+(z->at(4)-z-
->at(5)-z->at(6)-z->at(7))-(x->at(4)-x->at(5)-x->at(6)-x->at(7))+(z->at(4)+z->at(5)-z->at(6)+z->at(7))));

mpJ->setMatValue(2,0,(0.0625*((x->at(4)-x->at(5)-x->at(6)+x->at(7))+(y->at(4)+y-
->at(5)-y->at(6)-y->at(7))-(x->at(4)+x->at(5)-x->at(6)-x->at(7))+(y->at(4)-y->at(5)-y->at(6)-y->at(7))));

else if (face == 3) {
    mpJ->setMatValue(0,0,(0.0625*((y->at(0)-y->at(1)+y->at(4)-y->at(5))+(z->at(0)+z-
->at(1)-z->at(4)-z->at(5))-(y->at(0)+y->at(1)-y->at(4)-y->at(5))+(z->at(0)-z->at(1)+z->at(4)+z->at(5))));

mpJ->setMatValue(1,0,(0.0625*((x->at(0)+x->at(1)-x->at(4)+x->at(5))+(z->at(0)-z->at(1)+z->at(4)-z->at(5))-(x->at(0)-x->at(1)+x->at(4)+x->at(5))+(z->at(0)+z->at(1)-z->at(4)+z->at(5))));

mpJ->setMatValue(2,0,(0.0625*((x->at(0)-x->at(1)+x->at(4)-x->at(5))+(y->at(0)-y-
->at(1)-y->at(4)-y->at(5))-(x->at(0)+x->at(1)-x->at(4)-x->at(5))+(y->at(0)+y->at(1)-y->at(4)-y->at(5))));

else if (face == 4) {
    mpJ->setMatValue(0,0,(0.0625*((y->at(1)-y->at(2)-y->at(5)-y->at(6))+(z->at(1)+z-
->at(2)+z->at(5)+z->at(6))-y->at(1)+y->at(2)-y->at(5)-y->at(6))+(z->at(1)-z->at(2)-z-
->at(5)-z->at(6))));

mpJ->setMatValue(1,0,(0.0625*((x->at(1)+x->at(2)-x->at(5)-x->at(6))+(z->at(1)-z-
->at(2)+z->at(5)+z->at(6))-(x->at(1)-x->at(2)+x->at(5)-x->at(6))+(z->at(1)+z->at(2)-z-
->at(5)+z->at(6))));

mpJ->setMatValue(2,0,(0.0625*((x->at(1)-x->at(2)+x->at(5)-x->at(6))+(y->at(1)-y-
->at(2)-y->at(5)-y->at(6))-(x->at(1)+x->at(2)-x->at(5)+x->at(6))+(y->at(1)+y-
->at(2)+y->at(5)-y->at(6))));

else if (face == 5) {
    mpJ->setMatValue(0,0,(0.0625*((y->at(2)-y->at(3)+y->at(6)-y->at(7))+(z->at(2)+z-
->at(3)+z->at(6)+z->at(7))-(y->at(2)+y->at(3)-y->at(6)-y->at(7))+(z->at(2)-z-
->at(3)-z->at(6)-z->at(7))));

mpJ->setMatValue(1,0,(0.0625*((x->at(2)+x->at(3)-x->at(6)-x->at(7))+(z->at(2)-z-
->at(3)+z->at(6)-z->at(7))-(x->at(2)-x->at(3)+x->at(6)-x->at(7))+(z->at(2)+z-
->at(3)-z->at(6)+z->at(7))));

mpJ->setMatValue(2,0,(0.0625*((x->at(2)-x->at(3)+x->at(6)-x->at(7))+(y->at(2)+y-
->at(3)-y->at(6)-y->at(7))-(x->at(2)+x->at(3)-x->at(6)+x->at(7))+(y->at(2)+y-
->at(3)+y->at(6)-y->at(7))));

else if (face == 6) {
    mpJ->setMatValue(0,0,(0.0625*((y->at(0)-y->at(3)-y->at(4)-y->at(7))+(z->at(0)-z-
->at(3)+z->at(4)-z->at(7))-(y->at(0)+y->at(3)+y->at(4)-y->at(7))+(z->at(0)+z-
->at(3)-z->at(4)-z->at(7))));

mpJ->setMatValue(1,0,(0.0625*((x->at(0)-x->at(3)-x->at(6)-x->at(7))+(z->at(0)+z-
->at(3)+z->at(6)-z->at(7))-(x->at(0)+x->at(3)-x->at(6)-x->at(7))+(z->at(0)-z-
->at(3)-z->at(6)+z->at(7))));

mpJ->setMatValue(2,0,(0.0625*((x->at(0)+x->at(3)-x->at(6)-x->at(7))+(y->at(0)-y-
->at(3)+y->at(4)-y->at(7))-(x->at(0)-x->at(3)+x->at(4)-x->at(7))+(y->at(0)+y-
->at(3)-y->at(4)-y->at(7))));

}
```cpp
} else { std::cout << "Unknown face! C3D8::During getElemEqvNodalLoad" << std: :endl; }

//*****************************************************************************
for (int n=0; n<3; n++)
    mpO->setMatValue(n,n,pStressState->at(n));
mpO->setMatValue(1,0,pStressState->at(3));
mpO->setMatValue(2,0,pStressState->at(4));
mpO->setMatValue(0,1,pStressState->at(3));
mpO->setMatValue(2,1,pStressState->at(5));
mpO->setMatValue(0,2,pStressState->at(4));
mpO->setMatValue(1,2,pStressState->at(5));
mpR = mpN->multMat(mpO->multMat(mpJ));
for (int n=0; n<12; n++)
    eqvNodalLoad->at(n) = mpR->getMatValue(n,0);
//*****************************************************************************
delete x;
delete y;
delete z;
delete mpN;
delete mpJ;
delete mpR;
return eqvNodalLoad;
}

void C3D8::fillConstitutiveMat(Matrix * mpE)
{
    float v = mpMaterial->getPoisson();
    float E = mpMaterial->getYoungs();
    double c = E/((1+v)*(1-2*v));
    double G = E/(2*(1+v));
    //*****************************************************************************
    for (int n=0; n<3; n++)
        { mpE->setMatValue(n,n,(1-v)*c); }  
    for (int n=3; n<6; n++)
        { mpE->setMatValue(n,n,G); }  
    mpE->setMatValue(0,1,v*c);
    mpE->setMatValue(3,1,v*c);
    mpE->setMatValue(5,2,v*c);
    mpE->setMatValue(3,3,v*c);
    mpE->setMatValue(1,4,v*c);
    mpE->setMatValue(4,5,v*c);
    mpE->setMatValue(5,6,v*c);
    mpE->setMatValue(4,7,v*c);
    mpE->setMatValue(2,8,v*c);
}

void C3D8::fillStrainDisp(Matrix * mpA)
{
    mpA->setMatValue(0,0,1);
    mpA->setMatValue(3,1,1);
    mpA->setMatValue(5,2,1);
    mpA->setMatValue(3,3,1);
    mpA->setMatValue(1,4,1);
    mpA->setMatValue(4,5,1);
    mpA->setMatValue(5,6,1);
    mpA->setMatValue(4,7,1);
    mpA->setMatValue(2,8,1);
}

```
Source Code 20: matrix.h

```cpp
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <vector>
#include <math.h>

class Matrix
{

public:

    Matrix ( size_t, size_t );
~Matrix();

double getMatValue( size_t, size_t );
size_t getMatRows();
size_t getMatCols();
void setMatValue( size_t, size_t, double );
void displayMat();
void addMat( Matrix * );
Matrix * transposeMat();
Matrix * multMat( Matrix * );
double detMat();
Matrix * invMat();

private:

    size_t mRows, mCols;
   std::vector<double> mData;
};

#endif
```

Source Code 21: matrix.cpp

```cpp
#include "matrix.h"

Matrix::Matrix( size_t rows, size_t cols ) : mRows( rows ), mCols( cols ), mData( rows * cols ){}

Matrix::~Matrix()
{
    mData.clear();
}

double Matrix::getMatValue( size_t row, size_t col )
{
    return mData.at( row * mCols + col );
}

size_t Matrix::getMatRows()
{
    return mRows;
}
```
APPENDIX B. SOURCE CODE

size_t Matrix::getMatCols()
{       return mCols;
}

void Matrix::setMatValue( size_t row, size_t col, double value )
{       mData.at( row * mCols + col ) = value;
}

void Matrix::displayMat()
{       for ( unsigned int nRow = 0; nRow < mRows; nRow++ )
        {       for ( unsigned int nCol = 0; nCol < mCols; nCol++ )
                    {       std::cout << mData.at( nRow * mCols + nCol ) << " ";
                    }       std::cout << std::endl;
        }
}

void Matrix::addMat( Matrix *pMatB )
{       for ( unsigned int nRow = 0; nRow < mRows; nRow++ )
        {       for ( unsigned int nCol = 0; nCol < mCols; nCol++ )
                    {       mData.at( nRow * mCols + nCol ) = mData.at( nRow * mCols + nCol ) + pMatB->
                    getMatValue( nRow, nCol );
                    }
        }
}

Matrix * Matrix::transposeMat()
{       Matrix * pMatTrans = new Matrix( mCols, mRows );
        for ( unsigned int nRow = 0; nRow < mRows; nRow++ )
        {       for ( unsigned int nCol = 0; nCol < mCols; nCol++ )
                    {       pMatTrans->setMatValue( nCol, nRow, mData.at( nRow * mCols + nCol ) );
                    }       return pMatTrans;
        }
}

Matrix * Matrix::multMat( Matrix *pMatB )
{       Matrix * pMatMult;
        if ( mCols == pMatB->getMatRows() )
        {       pMatMult = new Matrix( mRows, pMatB->getMatCols() );
                double value;
                for ( unsigned int nRow = 0; nRow < pMatMult->getMatRows(); nRow++ )
                {       pMatMult->setMatValue( nCol, nRow, pMatMult->getMatCols() );
                }       return pMatMult;
        }
}
for ( unsigned int nCol = 0; nCol < pMatMult->getMatCols(); nCol++ )
{
    value = 0;
    for ( unsigned int nColA = 0; nColA < mCols; nColA++ )
    {
        value = value + ( getMatValue( nRow, nColA ) ) * ( pMatB->getMatValue( nColA, nCol ) );
    }
    pMatMult->setMatValue( nRow, nCol, value );
}
return pMatMult;
delte pMatMult;
}

else if ( pMatB->getMatRows() == 1 && pMatB->getMatCols() == 1 )
{
    pMatMult = new Matrix( mRows, mCols );
    for ( unsigned int nCol = 0; nCol < mRows; nCol++ )
    {
        for ( unsigned int nRow = 0; nRow < mCols; nRow++ )
        {
            pMatMult->setMatValue( nRow, nCol, getMatValue( nRow, nCol ) * pMatB->getMatValue( 0, 0 ) );
        }
    }
    return pMatMult;
delte pMatMult;
}
else
{
    std::cout << "ERROR! Wrong size of matrix, during Matrix::multMat()" << std::endl;
    return 0;
}

double Matrix::detMat()
{
    double detA = 0;
    double param = 0;

    if( mCols == 1 )
    {
        for( int n=0;n<mRows;n++)
        {
            param = param + getMatValue( n, 0 ) * getMatValue( n, 0 );
        }
        detA = sqrt( param );
    }
    else if( mRows != mCols )
    {
        std::cout << "ERROR! Not square matrix during Matrix::detMat()" << std::endl;
    }
    else if( mRows == 2 )
    {
detA = getMatValue( 0, 0 ) * getMatValue( 1, 1 ) - getMatValue( 0, 1 ) * getMatValue( 1, 0 );
}
else if( mRows == 3 )
{
    detA =
    ( ( getMatValue( 0, 0 ) * getMatValue( 1, 1 ) * getMatValue( 2, 2 ) ) +
    ( getMatValue( 0, 1 ) * getMatValue( 1, 2 ) * getMatValue( 2, 0 ) ) +
    ( getMatValue( 0, 2 ) * getMatValue( 1, 0 ) * getMatValue( 2, 1 ) ) -
    ( getMatValue( 0, 2 ) * getMatValue( 1, 1 ) * getMatValue( 2, 0 ) ) -
    ( getMatValue( 0, 1 ) * getMatValue( 1, 2 ) * getMatValue( 2, 1 ) ) ) ;
}
else
{
    std::cout << "ERROR! Unknown matrix during Matrix::detMat()" << std::endl;
}
return detA;
}

Matrix * Matrix::invMat()
{
    Matrix * pMatInv = new Matrix( mCols, mRows );
    if( mRows != mCols )
    {
        std::cout << "ERROR! Not square matrix during Matrix::invMat()" << std::endl;
        return 0;
    }
    else if( mRows == 2 )
    {
        Matrix * pMatInvDet = new Matrix( 1, 1 );
        pMatInvDet->setMatValue( 0, 0, 1/detMat() );
        pMatInv->setMatValue( 0, 0, getMatValue( 1, 1 ) );
        pMatInv->setMatValue( 1, 0, -getMatValue( 1, 0 ) );
        pMatInv->setMatValue( 0, 1, -getMatValue( 0, 1 ) );
        pMatInv->setMatValue( 1, 1, getMatValue( 0, 0 ) );
        Matrix * pMatInvFinal = pMatInv->multMat( pMatInvDet );
        delete pMatInv;
        delete pMatInvDet;
        return pMatInvFinal;
    }
    else if( mRows == 3 )
    {
        Matrix * pMatInvDet = new Matrix( 1, 1 );
        pMatInvDet->setMatValue( 0, 0, 1/detMat() );
        Matrix * pMatInv2x2 = new Matrix( 2, 2 );
        pMatInv2x2->setMatValue( 0, 0, getMatValue( 1, 1 ) );
        pMatInv2x2->setMatValue( 1, 0, getMatValue( 2, 1 ) );
        pMatInv2x2->setMatValue( 0, 1, getMatValue( 1, 2 ) );
        pMatInv2x2->setMatValue( 1, 1, getMatValue( 2, 2 ) );
        pMatInv->setMatValue( 0, 0, 0+pMatInv2x2->detMat() );
        pMatInv2x2->setMatValue( 0, 0, getMatValue( 1, 0 ) );
        pMatInv2x2->setMatValue( 1, 0, getMatValue( 2, 0 ) );
        pMatInv2x2->setMatValue( 0, 1, getMatValue( 1, 2 ) );
        pMatInv2x2->setMatValue( 1, 1, getMatValue( 2, 2 ) );
    }

pMatInv->setMatValue( 1, 0, 0-pMatInv2x2->detMat() );
pMatInv2x2->setMatValue( 1, 0, getMatValue( 1, 0 ) );
pMatInv2x2->setMatValue( 0, 1, getMatValue( 1, 1 ) );
pMatInv2x2->setMatValue( 1, 1, getMatValue( 1, 1 ) );
pMatInv->setMatValue( 2, 0, 0+pMatInv2x2->detMat() );
pMatInv2x2->setMatValue( 0, 0, getMatValue( 0, 1 ) );
pMatInv2x2->setMatValue( 1, 0, getMatValue( 2, 1 ) );
pMatInv2x2->setMatValue( 0, 1, getMatValue( 0, 2 ) );
pMatInv2x2->setMatValue( 1, 1, getMatValue( 2, 2 ) );
pMatInv->setMatValue( 0, 1, 0-pMatInv2x2->detMat() );
pMatInv2x2->setMatValue( 0, 0, getMatValue( 0, 0 ) );
pMatInv2x2->setMatValue( 1, 0, getMatValue( 2, 0 ) );
pMatInv2x2->setMatValue( 0, 1, getMatValue( 0, 1 ) );
pMatInv2x2->setMatValue( 1, 1, getMatValue( 2, 1 ) );
pMatInv->setMatValue( 2, 1, 0-pMatInv2x2->detMat() );
pMatInv2x2->setMatValue( 0, 0, getMatValue( 0, 1 ) );
pMatInv2x2->setMatValue( 1, 0, getMatValue( 1, 1 ) );
pMatInv2x2->setMatValue( 0, 1, getMatValue( 0, 2 ) );
pMatInv2x2->setMatValue( 1, 1, getMatValue( 1, 2 ) );
pMatInv->setMatValue( 0, 2, 0+pMatInv2x2->detMat() );
pMatInv2x2->setMatValue( 0, 0, getMatValue( 0, 0 ) );
pMatInv2x2->setMatValue( 1, 0, getMatValue( 1, 0 ) );
pMatInv2x2->setMatValue( 0, 1, getMatValue( 0, 2 ) );
pMatInv2x2->setMatValue( 1, 1, getMatValue( 1, 2 ) );
pMatInv->setMatValue( 1, 2, 0-pMatInv2x2->detMat() );
pMatInv2x2->setMatValue( 0, 0, getMatValue( 0, 0 ) );
pMatInv2x2->setMatValue( 1, 0, getMatValue( 1, 0 ) );
pMatInv2x2->setMatValue( 0, 1, getMatValue( 0, 1 ) );
pMatInv2x2->setMatValue( 1, 1, getMatValue( 1, 1 ) );
pMatInv->setMatValue( 2, 2, 0+pMatInv2x2->detMat() );
Matrix * pMatInvFinal = pMatInv->multMat( pMatInvDet );
delete pMatInv;
delete pMatInvDet;
delete pMatInv2x2;
return pMatInvFinal;

else
{
    std::cout << "ERROR! Unknown matrix during Matrix::invMat" << std::endl;
    return 0;
}
}