TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

B-12-BME-2018/2023

USING DEAL.II FOR STRUCTURAL ANALYSIS OF A MEDICAL OXYGEN CYLINDER

BY:
HIMAL KUMAR RANA MAGAR (075BME018)
KHIM BAHADUR BASNET (075BME022)
LILANATH GHIMIRE (075BME024)

A PROJECT REPORT
SUBMITTED TO DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR DEGREE OF BACHELOR IN MECHANICAL ENGINEERING

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING
LALITPUR, NEPAL

MARCH 2023

COPYRIGHT

Associate Prof. Surya Prasad Adhikari, PhD
Head of Department
Department of Mechanical and Aerospace Engineering
Pulchowk Campus, Institute of Engineering
Pulchowk, Lalitpur
Nepal

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

DEPARTMENT OF MECHANICAL AND AEROSPACE ENGINEERING

The undersigned certify that they have read, and recommended to the Institute of Engineering for acceptance, a project report entitled "Using deal.ii for structural analysis of a Medical Oxygen Cylinder"submitted by Himal Kumar Rana Magar, Khim Bahadur Basnet and Lilanath Ghimire in partial fulfillment of the requirements for the degree of Bachelor of Mechanical Engineering.

Supervisor, Dr. Mahesh Chandra Luintel
Professor
Department of Mechanical and Aerospace Engineering

Supervisor, Kamal Darlami
Assistant Professor
Department of Mechanical and Aerospace Engineering

External Examiner, Janak Kumar Tharu
Assistant Professor
Nepal Engineering College

Committee Chairperson, Dr. Surya Prasad Adhakari
Head of Department
Department of Mechanical and Aerospace Engineering

Date

ABSTRACT

Medical Oxygen Cylinders being convenient containers for transportation and storage of oxygen gas has wide range of applications in industries and hospitals. Storing oxygen at high pressure in cylinders increases the risk of structural failure. This study aims to develop a program based on open source C++ library for Finite Element Analysis (FEA) in thin-walled cylinders and to apply the program to perform structural analysis and safety assessment of medical oxygen cylinders operating below their working pressure. To achieve this goal, a literature review was conducted to identify gaps in the use of open-source software for FEA in thin-walled cylinders. Open source FEA library called deal.ii was used to construct the program. This C++ based program was verified by comparing it with the hoop stress results from theoretical calculation and ANSYS for a simple hollow cylinder. The study then proceeded to create a detailed CAD model of the medical oxygen cylinder for a parametric study for varying wall thickness and material. The model was simplified and meshed for FEA, with pressure and fixity constraints applied during simulation. The parametric simulations were run through the developed program which showed that maximum hoop stress occurs in the region around neck of the cylinder. The methodology involved verification of the program by comparing the results with that of ANSYS and theoretical calculations in case of simple hollow cylinder in which maximum error was found to be 0.875 %. Parametric analysis for variying material and thickness found that the cylinder with larger thickness i.e., 5.6 mm, and material 37MnSi5 undergoes through the smallest deformation. The development of the open-source software will provide a valuable resource for future research and development in this field, as well as contribute to the enhancement of the safety and reliability of medical oxygen cylinders.

Keywords: Open-source library, Finite Element Analysis, deal.ii, Medical Oxygen Cylinder, Structural analysis

ACKNOWLEDGEMENT

We would like to express our deepest gratitude to the Department of Mechanical and Aerospace Engineering at IOE, Pulchowk Campus, Lalitpur, for providing us with the invaluable opportunity to work on a project that allowed us to apply and expand upon the knowledge we gained during our Bachelor's program in Mechanical Engineering. Our sincere thanks go to Dr. Surya Prasad Adhikari, Head of Department, Department of Mechanical and Aerospace Engineering; Assistant Prof. Lakshman Motra, Deputy Head of Department, Department of Mechanical and Aerospace Engineering; and Yashoda Adhikari, Administrator, Department of Mechanical and Aerospace Engineering, for their unwavering support throughout the project. Their guidance, resources, and suggestions were instrumental in making our project successful and meaningful.

We would also like to express our heartfelt appreciation to our supervisors, Professor Dr. Mahesh Chandra Luitel and Assistant Professor Kamal Darlami, for their invaluable insights and guidance throughout the project. Their expertise and encouragement helped us navigate through the various phases of the project, and we learned a great deal from them. Moreover, we would like to offer our sincere acknowledgement to Mr. Dipak Ghimire of Sagarmatha Oxygen Pvt. Ltd. and Surendra K.C. of Kantipur Oxygen Pvt. Ltd. for their invaluable insights on the information needed for our project. Their expertise and support were critical in helping us to understand the technical aspects of our project, and we are grateful for their assistance. Finally, we would like to convey our thanks to all the esteemed teachers, seniors, juniors, and peers who supported us throughout this project. Their encouragement and assistance were invaluable, and we are deeply grateful for their contributions to our success.

Himal Kumar Rana Magar (075BME018)
Khim Bahadur Basnet (075BME022)
Lilanath Ghimire (075BME024)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

$\boldsymbol{\sigma}$	stress tensor,
$\vec{b}$	body force vector,
$\vec{t}$	prescribed traction vector,
$\vec{d}$	prescribed displacement vector,
$\vec{u}$	displacement vector,
$\vec{n}$	normal vector to the surface,
Ω	open domain,
Γ_t	boundary surface when traction is prescribed,
Γ_d	boundary surface when displacement is prescribed,
$\bar{\Omega}$	closed domain, i.e. $\bar{\Omega} = \Omega \cup \Gamma_t \cup \Gamma_d$
ϕ	shape function
$\mathbf{H}$	Hilbert space
λ	Lame Modulus,
μ	Modulus of Rigidity/Shear Modulus,
ν	Poisson's Ratio,
E	Young,s Modulus of Elasticity,
δ_{ij}	Kronecker delta operator

LIST OF ACRONYMS AND ABBREVIATIONS

DEAL	Differential Equation Analysis Library
FEA	Finite Element Analysis
FEM	Finite Element Method
MOC	Medical Oxygen Cylinder,
MPI	Message Passing Interface

CHAPTER ONE: INTRODUCTION

1.1 Background

Oxygen cylinders are high pressure thin-walled container used for various purposes. Mostly, these cylinders are used in medical sector. Typically, it contains oxygen at a pressure around 150 bar. Various sizes of oxygen cylinders are available in the market, ranging from 10 to 50 liters in capacity.

Since the discovery of oxygen, scientists and technicians have been continuously exploring and developing various technologies to produce and transport this essential gas. One such technology that has proved to be highly effective is the medical oxygen cylinder. The main function of this cylinder is to safely and affordably transport oxygen from manufacturing plants to hospitals and other medical facilities. In contrast to longer pipelines, oxygen cylinders provide a more convenient and practical solution for delivering oxygen where it is needed.

Oxygen cylinders have been designed to meet the specific requirements of medical facilities and other industries. They are manufactured using high-strength steel alloys or lightweight aluminum materials, which offer excellent strength and durability. Oxygen cylinders are available in various sizes and shapes, depending on the specific application, and can be filled with compressed oxygen gas up to a working pressure of 150Bar.

The initial development of the oxygen cylinder was around 1868 AD. As the use of oxygen increased in fields such as medicine and industry, modifications were proposed to increase the cylinder's capacity to carry oxygen at high pressure, while reducing production costs and allowing for mass production. Today, there are several types of oxygen cylinders available with similar physical appearances but varying in dimensions and materials depending on their purpose and use. Typically, oxygen cylinders are constructed using steel alloys or aluminum.

Oxygen cylinders have several uses, which include:

- Providing respiratory support in medical facilities
- Assisting with breathing in high-altitude environments
- Supporting diving activities
- Administering oxygen therapy
- Facilitating industrial processes like welding, lamp-working, and gas cutting.

On another note, Finite Element Analysis (FEA) is used extensively in engineering and scientific fields to simulate and analyze the behavior of complex systems and structures

under various conditions. These software tools are used to predict how a particular design will perform under various loads, stresses, and strains, and can help engineers optimize designs, reduce costs, and improve performance.

The need for open source FEA software has become increasingly important in recent years, as the cost of proprietary software can be prohibitively high for small businesses, individual users, and academic institutions. Open source FEA software provides an alternative that is accessible to a wider range of users, regardless of their financial resources.

In addition to cost considerations, open source FEA software also provides benefits in terms of transparency and collaboration. With open source software, users can access the source code and modify it to suit their needs. This level of transparency also promotes collaboration and sharing among users, leading to a more robust and diverse community of developers and users.

Moreover, open source FEA software also allows for greater customization and integration with other software tools, which can improve productivity and workflow efficiency. Users can develop their own plugins or interfaces to integrate FEA software with other design or analysis tools, leading to a more streamlined design process.

Overall, the need for open source FEA software is driven by the desire for greater accessibility, transparency, collaboration, and customization in the engineering and scientific community. By leveraging the power of open source software, users can access powerful simulation and analysis tools that can help them optimize their designs, reduce costs, and improve performance.

The aim of this project is to perform a structural analysis of a medical oxygen pressure cylinder using open source platform called deal.ii. It is a large finite element library written in C++ that offers numerous capabilities to work in FEA. The goal is to determine the level of safety when the cylinder is used below its working pressure. Additionally, this project aims to provide suggestions for minimizing the risk of deformation and failure of the oxygen cylinder under low-pressure conditions. The study will focus on a typical D-type 46.7L oxygen cylinder with a working pressure of 150 bar. By conducting this analysis, It was expected to provide valuable insights that can be used to enhance the safety and reliability of oxygen cylinders in medical and other fields.

Figure 1.1: Typical Medical Oxygen Cylinder

Note: The Picture in figure 1.1 was taken in Sagarmatha Oxygen Pvt.Ltd during project field visit,it is of capacity 46.7L, More about shape, size and dimensions of oxygen cylinder is explained in Field Visit section of this report under chapter Methodology. Also, the terms displacement and deformation are used interchangeably throughout the report.

1.2 Problem Statement

Storing oxygen at high pressures in cylinders increases the risk of structural failure. The pressure can cause the cylinder to explode, leak or crack, and oxygen's chemical properties increase the likelihood of corrosion if the cylinder is filled without checking for the presence of water/humidity or an electrolytic environment on the internal surface of the cylinder. During a visit to an oxygen filling industry, it was learned that recently a oxygen cylinder failure resulted in the loss of two lives. Unfortunately, news of oxygen cylinder explosions resulting in death or serious injury has become increasingly common. Therefore, to ensure the safety of using oxygen cylinders at their maximum working pressure and temperature, a safety analysis is necessary. Predicting failure based on general knowledge or common prediction methods is challenging. Hence, this project aims to analyze the bursting failure of oxygen cylinders due to excessive pressure, determine the safety state of oxygen cylinders at their working pressure, and analyze the use of different materials and their resulting safety. The potential hazards of oxygen cylinders require that safety be a top priority, especially in the medical industry, where patients depend on them for life support. Understanding the safety limits of oxygen cylinders is essential to avoid accidents and fatalities. Therefore, this project aims to develop an open-source C++ code to predict the safety and risk of oxygen cylinders operating below their working pressure. By analyzing the structural integrity of the D-type 46.7L typical oxygen cylinder with a working pressure of 150 bar, this project aims to provide useful suggestions for minimizing the deformation and risk of failure of oxygen cylinders below their working pressure. Furthermore, the project seeks to explore the effectiveness of different materials in enhancing the safety of oxygen cylinders. Through this analysis,it is hoped that it contributes to the development of safer and more reliable oxygen cylinders for various applications.

Figure 1.2: Bursted Oxygen Cylinder
(source:safetymattersweekly.com)

1.3 Objectives

1.3.1 Main Objective

To perform structural analysis and safety assessment of medical oxygen cylinder.

1.3.2 Specific Objectives

- To develop C++ code that can accurately predict the safety and risk of medical oxygen cylinders operating below their working pressure.
- To investigate the effectiveness of different materials and thicknesses in enhancing the safety of medical oxygen cylinders.
- To Compare and validate the simulation results with theoretical calculations to ensure the accuracy and reliability of the model.

CHAPTER TWO: LITERATURE REVIEW

2.1 Theoretical Background

Followings are the theories behind the working of the program and the model:

2.1.1 Mathematical Formulation

Strong Form

The strong form for the small displacement three-dimensional linear elasticity problem with Neumann and Dirichlet boundary conditions is,
Given $\vec{b} : \Omega \to \mathbb{R}^3,\ \vec{d} : \Gamma_d \to \mathbb{R}^3,\ \vec{t} : \Gamma_t \to \mathbb{R}^3$, find $\vec{u} : \bar{\Omega} \to \mathbb{R}^3$ such that,

$$\begin{aligned} -\text{div}(\boldsymbol{\sigma}(\vec{u})) &= \vec{b} \text{ in } \Omega \\ \vec{u} &= \vec{d} \text{ on } \Gamma_d \\ \boldsymbol{\sigma}(\vec{u}) \cdot \vec{n} &= \vec{t} \text{ on } \Gamma_t \end{aligned}$$

Weak Form

Weak form of the problem in bi-linear form is,
Given $\vec{b} : \Omega \to \mathbb{R}^3,\ \vec{d} : \Gamma_d \to \mathbb{R}^3,\ \vec{t} : \Gamma_t \to \mathbb{R}^3$, find $\vec{u} : \bar{\Omega} \to \mathbb{R}^3$ such that for $\vec{v} \in \mathbf{H}$,

$$a(\vec{u}, \vec{v}) = (\vec{t}, \vec{v})_{\Gamma_t} + (\vec{b}, \vec{v}) \text{ in } \Omega$$

where,
$a(\vec{u}, \vec{v}) = \int_\Omega \boldsymbol{\sigma}(\vec{u}) : \nabla \vec{v}$
$(\vec{t}, \vec{v})_{\Gamma_t} = \int_{\Gamma_t} \vec{t} \cdot \vec{v}$
$(\vec{b}, \vec{v}) = \int_\Omega \vec{b} \cdot \vec{v}$
Note:
Stress tensor for isotropic material is given by,

$$\boldsymbol{\sigma}(\vec{u}) = \lambda \, (\text{div } \vec{u}) \, \boldsymbol{I} + 2\mu\varepsilon(\vec{u})$$

where,
λ and μ are the Lame parameters, $\boldsymbol{I}$ is the second rank identity tensor, and
$\varepsilon(\bullet) := \{\nabla(\bullet) + \nabla(\bullet)^T\}/2$.
The meaning of each symbols are listed on LIST OF SYMBOLS chapter.

Finite Element Approximation

Applying finite element approximation, the problem can be expressed as the linear system of equations and is represented as,

AU=F

where,
A is the global stiffness matrix,
U is the global displacement vector,
F is the global force vector.
Global stiffness matrix is defined as,

$$A_{ij} = \sum_{k,l} \{(\lambda \partial_l(\Phi_i)_l, \partial_k(\Phi_j)_k)_\Omega + (\mu \partial_k(\Phi_i)_l, \partial_k(\Phi_j)_l)_\Omega + (\mu \partial_k(\Phi_i)_l, \partial_l(\Phi_j)_k)_\Omega\}$$

here, i and j run over the global degrees of freedom while k and l run over the space-dimension, and Φ_i represents the vector shape function, which is defined as,

$$\Phi_i(\mathbf{x}) = \phi_i(\mathbf{x})\ \mathbf{e}_{\text{comp}(i)}$$

where, **e** is the unit vector specified by comp(i) which in turn is defined as,

$$\text{comp}(i) = \begin{cases} 0 \text{ if } i = 0, 3, 6, \cdots \\ 1 \text{ if } i = 1, 4, 7, \cdots \\ 2 \text{ if } i = 2, 5, 8, \cdots \end{cases}$$

The global stiffness matrix is obtained by assembling the local stiffness matrices, where local stiffness matrix on cell K is expressed as,

$$A_{ij}^K = \sum_{k,l} \{(\lambda \partial_l(\Phi_i)_l, \partial_k(\Phi_j)_k)_K + (\mu \partial_k(\Phi_i)_l, \partial_k(\Phi_j)_l)_K + (\mu \partial_k(\Phi_i)_l, \partial_l(\Phi_j)_k)_K\}$$

This can be further arranged into,

$$A_{ij}^K = \left(\lambda \partial_{\text{comp}(i)}\phi_i, \partial_{\text{comp}(j)}\phi_j\right)_K + \left(\mu \partial_{\text{comp}(j)}\phi_i, \partial_{\text{comp}(i)}\phi_j\right)_K + (\mu \nabla\phi_i, \nabla\phi_j)_K\, \delta_{\text{comp}(i),\text{comp}(j)}$$

here, now i and j run over local degrees of freedom, and δ_{ij} is the Kronecker Delta operator.
Since, integration on computer is difficult to incorporate and sometimes even impossible, we use Gauss Quadrature formula. For this, we transform the cells from real(physical) space into parametric space and vice versa using **Jacobian** transformation, and then apply Gauss Quadrature formula to compute the integration.

Then, final expression becomes,

$$A_{ij}^K = \sum_q \{\lambda \left(\partial_{\text{comp}(i)}\phi_i(q)\right)\left(\partial_{\text{comp}(j)}\phi_j(q)\right) + \mu \left(\partial_{\text{comp}(j)}\phi_i(q)\right)\left(\partial_{\text{comp}(i)}\phi_j(q)\right) + \mu \left(\nabla\phi_i(q)\right)\left(\nabla\phi_j(q)\right)\delta_{\text{comp}(i),\text{comp}(j)}\}\text{JxW}(q)$$

where, $q's$ represent quadrature points in parametric space, and 'JxW(q)' represents the product of determinant of Jacobian and the weight at the quadrature point q.
Similarly, right-hand-side vector(global force vector) is obtained by assembling the contributions from the local force vectors, which are defined as,

$$F_i^K = \sum_{q_f} \left(t_{\text{comp}(i)}\phi_i(q_f)\right)\text{JxW}(q_f) + \sum_q \left(b_{\text{comp}(i)}\phi_i(q)\right)\text{JxW}(q)$$

where q_f refers to the quadrature points on the face of the cell, since the first term is to be integrated on the face of the cell, that belongs to the boundary surface Γ_t.
In the problem, traction is caused by the gas pressure, i.e.,

$$\vec{t} = -P \cdot \vec{n}$$

where, P is the pressure acting on the surface and $\vec{n}$ is the normal surface vector of the surface.
Then,

$$\sum_{q_f} \left(t_{\text{comp}(i)}\phi_i(q_f)\right)\text{JxW}(q_f) \;=\; \sum_{q_f} \left(-P \cdot n_{\text{comp}(i)}\phi_i(q_f)\right)\text{JxW}(q_f)$$

Finally, the rhs vector on a local cell becomes,

$$F_i^K = \sum_{q_f} \left(-P \cdot n_{\text{comp}(i)}\phi_i(q_f)\right)\text{JxW}(q_f) + \sum_q \left(b_{\text{comp}(i)}\phi_i(q)\right)\text{JxW}(q)$$

The complete derivation to these expressions is present in the Appendix A and is heavily based on the book by Ioannis Koutromanos and Roy (2018) and Chaves (2013).

2.1.2 Thin-Walled Cylinder

In the case of thin-walled cylinders, the relationship between the wall thickness and diameter is often expressed as D/t » 1, where D/t typically exceeds 20. Due to the radially symmetric geometry and loading of cylindrical vessels, the stresses within them do not vary in the angular direction. The impact of end caps can be ignored at locations far enough from them. Additionally, as the wall thickness is insignificant, radial stress is considered absent. The two types of stress that exist in a pressure-filled thin-walled cylinder are hoop

stress and longitudinal stress.

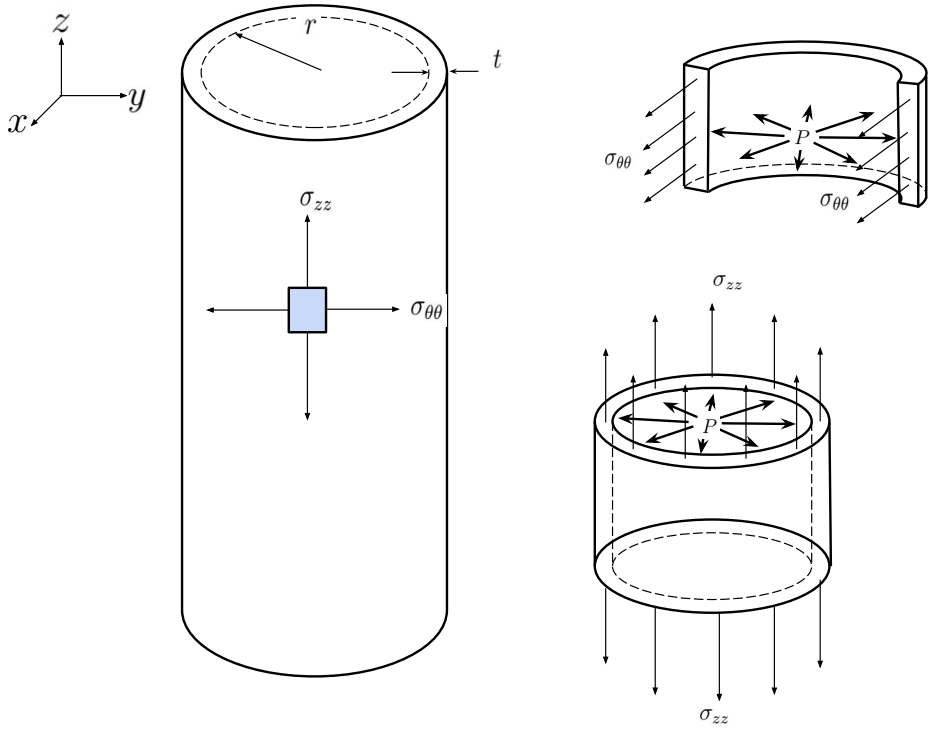


Figure 2.1: Thin-walled cylinder

The Hoop stress, $\boldsymbol{\sigma}_\theta$ and axial stress,$\boldsymbol{\sigma}_z$ acting on the cylinder are given by,

$$\begin{aligned} \boldsymbol{\sigma}_{\theta\theta} &= \frac{P \cdot r}{t} \\ \boldsymbol{\sigma}_{zz} &= \frac{P \cdot r}{2t} \end{aligned}$$

thus, hoop stress is twice the axial stress in case of thin wall cylinder in case of closed end cylinder, i.e., $\boldsymbol{\sigma}_{\theta\theta} = 2\boldsymbol{\sigma}_{zz}$.

2.2 FEA of Thin-Walled Cylinders

Thin-walled cylinders are widely used in various engineering applications, such as pressure vessels, storage tanks, pipelines, and aerospace structures. The structural behavior of thin-walled cylinders is complex and depends on factors such as the geometry, material properties, loading conditions, and boundary conditions. Finite Element Analysis (FEA) is a powerful tool for analyzing the stress, deformation, and failure of thin-walled cylinders. In recent years, there has been an increasing interest in the FEA of thin-walled cylinders due to its potential for improving the design, optimization, and safety of engineering structures. In this literature review, we will survey and analyze the existing research on the FEA of thin-walled cylinders, focusing on the key challenges, methodologies, and applications.

In a study done by Abdussalam (2006), FEA of the design and manufacture of aerosol cans was done. The author prefers the FEA over traditional "Design-by-test" methods because of repeatability and rapid re-analysis capacity.

Finite element analysis has also been used to investigate catastrophic failures of thin-walled cylinders. Mirzaei (2008) discussed the finite element simulations of deformation and fracture of a gas cylinder that catastrophically failed as a result of an accidental explosion. The FEA results clearly showed that the stresses caused by the assumed loading profile were indeed capable of creating local ruptures at the actual crack initiation sites.

Major of the studies involve comparison of FEA result with theoretical calculations and experimental data. A study performed Finite element analysis of specified thick wall cylinder with help of ANSYS software and compared its result with experimental result and theoretical calculation by Lame's equation (Macwan et al., 2011). They measured hoop strain and hoop stress by experimental setup for internal as well as external surface of cylinder and by numerical and theoretical method they calculated hoop and radial stress and concluded that there is about one percentage error between results of theoretical calculation and numerical method, 3.33 percentage error between theoretical calculation and experiment result and, 4 percentage error between result of experiment and numerical solution.

Rangari (2012) performed finite element analysis of LPG cylinder to verify its burst pressure. In this research the researchers assumed an LPG cylinder of material low carbon steel and calculated maximum shear stress, equivalent shear stress at critical area of failure by FEA on ANSYS Workbench as well as by using theoretical calculation and compared results. They concluded the verification of ANSYS result with theoretical result.

Wang et al. (2017) studied the buckling behavior of tori-spherical bottom head of a residential water heater tank. Both FEA and Hydrostatic test results were correlated to find that the effect of geometric imperfection has more effect on buckling pressure than contact imperfection.

P. Palanivelu (2017) performed Finite element analysis on a typical pressure vessel with ellipsoidal head to determined stress distribution and critical points of possible failure and result compared with theoretical calculation. The research found that equator of head of pressure vessel is critical point for failure.

Mohamed (2018) showed that the finite element method can give results with good agreement with the criteria of mechanics of material. Nevertheless, the model was of thin-walled cylinder and simplified geometry, it clearly shows the distribution of hoop and longitudinal stresses over the cylinder thickness.

Yin et al. (2019) conducted FEA analysis using ANSYS workbench to analyze a 40L industrial gas cylinder and found the maximum stresses are near to the allowable stresses. The authors also used equivalent linearization method to optimize the cylinder structure. It was found that these methods significantly improve the safety of the cylinder transportation process.

Das and Islam (2019) compared the deformation and stress distribution for continuous, discontinuous and material interface joint with help of FEA result for a thick wall pressure vessel and found that effect of geometry discontinuity is quite significant in von-misses and hoop stress.

Nendra Wibawa et al. (2021) did FEA for thick-walled cylinder for rocket motor case, in which they performed FE simulation for different wall thickness of thick-walled cylinder with same length and same outer diameter also for every increasing thickness they increased internal pressure for three different material Aluminum 6061, CFRP, GFRP and concluded that maximum hoop and longitudinal stress decreases for increase in wall thickness for given reference rocket motor casing.

FEA with nonlinear stabilization techniques and failure criterion can determine the burst pressures of the thin-walled cylinders accurately, while the conventional elastic-strain hardening plasticity material model may overestimate the burst pressure of a cylinder composed of plain carbon steel with a yield plateau(Wang et al., 2021).

2.3 Deal.ii

Finite Element Analysis (FEA) of thin-walled cylinders involves complex mathematical models and requires high-performance computing resources. Open-source software provides a cost-effective and customizable solution for FEA, as it enables users to access and modify the source code for their specific needs. Deal.ii (Arndt et al., 2022) is a C++ software library supporting the creation of finite element codes and an open community of users and developers. Deal.ii is a powerful open-source finite element software package that provides a wide range of tools and capabilities for FEA of various types of problems. It has strong focus on high-performance and parallel computing, which is essential for efficiently solving large-scale FEA problems.

Bangerth et al. (2007) in their paper, provide an overview of the deal.ii library, including its design principles, basic usage, and capabilities. It also discusses some of the advanced features of deal.ii, such as support for adaptive mesh refinement and parallel computing. This paper (Kronbichler & Kormann, 2012) describes the development of a generic interface for parallel computing in deal.ii. The authors demonstrate the effectiveness of their approach using a variety of test cases, including the solution of the Navier-Stokes equations.

This paper (Arndt et al., 2021a) provides an overview of the new features and improvements in deal.ii release 9.0. The authors discuss enhancements to the finite element spaces supported by deal.ii, improvements in the parallel computing capabilities, and updates to the interface for mesh generation (Arndt et al., 2021b)provides an overview of the parallel capabilities of deal.ii and its applications in solving elasticity equations. The authors discuss the parallel algorithms used in deal.ii, including domain decomposition and shared memory parallelization, and demonstrate their effectiveness in solving a range of elasticity problems.

2.4 Research gap

The literature review on Finite Element Analysis (FEA) in thin-walled cylinders reveals that the use of open source software for this application is limited. While there is existing research on FEA in thin-walled cylinders, the majority of studies have utilized commercial software. Therefore, a research gap exists regarding the application of open source software for FEA in thin-walled cylinders. Further investigation is required to explore the potential benefits and limitations of utilizing open source software for this application, and to determine the accuracy and reliability of results obtained from such software.

CHAPTER THREE: METHODOLOGY

This project involved designing and simulating a Medical oxygen cylinder to assess its safety and strength conditioning under different materials and geometric properties (thickness).The methodological flow chart of this project is shown in figure 3.1 .The project began with a literature review and field visits to understand the different types and specifications of oxygen cylinders available in the local as well as global market. Using this information, a detailed CAD model of the cylinder was created in SOLIDWORKS, including multiple versions with varying wall thicknesses for the simulation and analysis. The model was later simplified and meshed for finite element analysis, with pressure and fixity constraints applied during simulation. The simulations were run through the deal.ii library based C++ code and verified using theoretical calculations and ANSYS simulations.furthermore for entire course of this project literature review was conducted on related topics .

In conclusion, the project showcases the importance of a comprehensive and data-driven approach to designing, simulating and analyzing engineering systems. By using a range of methods, including literature reviews, field visits, CAD modeling, meshing, parametrization, coding and simulations,This project work provided a sound understanding of analysis under different study parameters.This projects highlighted the value of collaboration between different engineering disciplines, as this project drew on knowledge from multiple areas, including mechanical engineering, materials science, and computational modeling. The resulting oxygen cylinder analysis results are expected to have a significant positive impact on construction of upcoming oxygen bottles, providing a reliable and safe oxygen cylinder for patients as well as for industrial purposes.

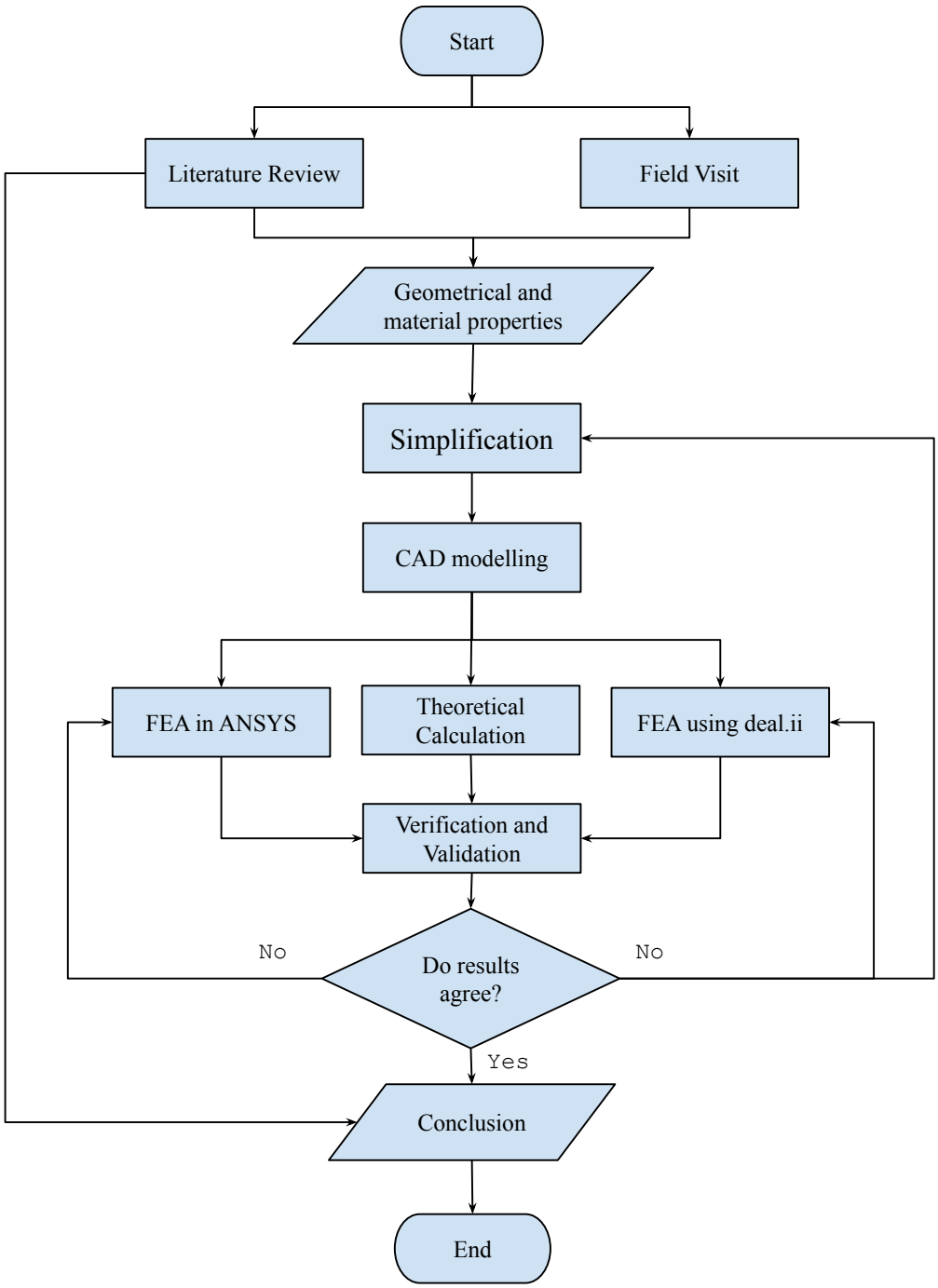


Figure 3.1: Methodology Flowchart

Each stage of the project work is explained in the respective sections.

3.1 Literature Review

Throughout the duration of the project, a comprehensive literature review was conducted to ensure that, the standard pathway were follower for project. Literature review was conducted to consult and analyze existing research and theories in order to compare the output of each step with those of established methods. This process was to ensure that the project was rigorous, thorough, and aligned with established best practices in the field. By conducting a careful and thorough literature review, to create a project upon existing knowledge that was grounded in solid research and theory.

3.2 Field Visit

During the starting of this project, field visits were conducted to two of the re-known oxygen gas manufacturing and refilling industries in Nepal to gain a better understanding of the different dimensions, capacities, and materials of oxygen cylinders that are available in the local market. These visits allowed to observe firsthand the different types of cylinders that were being used and imported from India and China. The carefully documentation of findings from these field visits were done and, which are presented in below. This information includes a summary of the different types of cylinders that were observed, their respective dimensions and capacities, and the materials used in their construction.

1. Sagarmatha Oxygen Pvt. Ltd (Patan Industrial State, Lalitpur)
    - Visited: 2079/04/24
    - Available Sizes (10L, 20L, 47L i.e. 46.7L & 50L)
    - Outer Diameter (OD)=232mm
    - Height(H) =1370mm
    - Working Pressure=150bar
    - Circumference of Neck: Upper (c') =25cm, Lower (c") =35cm
    - Upper Neck Height (H_1) =20mm
    - Lower Neck Height (H_2) =45mm
    - Height to Head Start (H_n) =1235mm
    - Head Profile: Spherical
2. Kantipur Oxygen Limited (Harsiddhi, Lalitpur)
    - Visited: 2079/04/24
    - OD =232mm
    - H= 1350mm

- Pressure =150bar
- H_1=3cm
- H_2=6cm
- C'=25cm
- C''=35cm
- Hole Dia. =3mm
- Thickness (t)=5.4mm
- Weight (W)=50.7kg
- Head Profile: Spherical

The above geometrical parameters are as shown in figure:

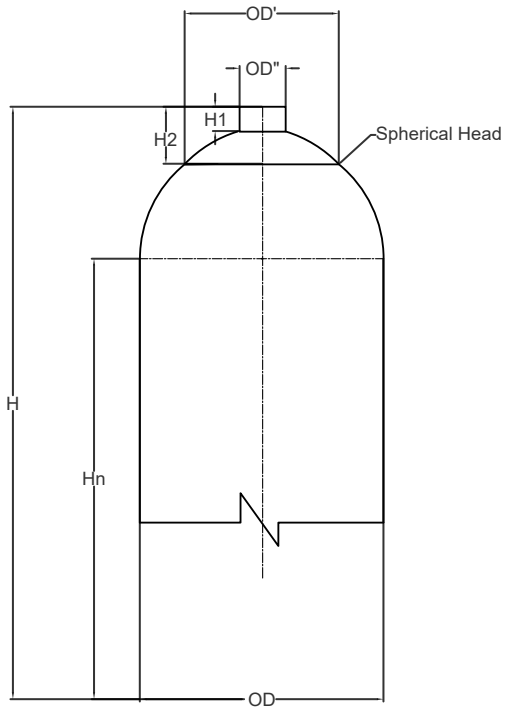


Figure 3.2: Body And Head Profile

3. Model finalized for simulation
    - OD=232mm
    - H=1365mm
    - Thickness(t)=5.6mm/5.5mm/5.4mm
    - Material = 32CrMo4/37MnSi5/34Mn2V

Following are some of the pictures taken during our project field:

Figure 3.3: Picture of Body And Head Profile

Figure 3.4: Valve in Head

Figure 3.5: Cylinder Bottom Profile

3.3 Geometrical and material properties

Based on preliminary field study, literature review, and analysis of design standards and codes,the necessary design geometry for a Medical oxygen cylinder with a capacity of 46.7L (internal volume) was obtained. This involved carefully considering the appropriate dimensions and specifications for the cylinder, including its diameter, height, and thickness, as well as other critical design elements. With this design geometry in hand, further calculations, processing ,refinement and simulation were performed and our that ensured the design met all necessary standards and requirements. This included analyzing the strength and safety of the cylinder, as well as considering factors such as weight, portability, and ease of use. By using a rigorous and data-driven approach, we were able to design and analyse a high-quality and effective medical oxygen cylinder that met all necessary specifications and requirements and which can be used in our deal.ii based simulation.

In selecting the appropriate geometry for this study, thorough review of the literature and field visit data was conducted. After careful consideration, the geometry proposed in (Yin et al., 2019) was chosen based on its suitability for research question of this project and its demonstrated effectiveness in previous studies. The studied cylinder geometry is of the following dimensions:

Outer Diameter	Wall Thickness	Height
232 mm	5.4 mm 5.5 mm 5.6 mm	1365 mm

Table 3.1: Dimensions of Medical Oxygen cylinder.

On literature review, the materials used in manufacture of high pressure gas cylinder are majorly 34Mn2V (Yin et al., 2019), 34CrMO4 (Bultel & Vogt, 2010; Li et al., 2019) and 37MnSi5. The material properties are shown in the table 3.2:

Material	Modulus of Elasticity (GPa)	Poisson's ratio	density (Kg/m^3)
34Mn2V	185	0.3	7850
34CrMo4	197	0.29	7850
37MnSi5	206	0.3	7850

Table 3.2: Materials and their properties.

The formulae to calculate the Lame's parameters out of **E** and ν are:

$$\mu = G = \frac{E}{2(1+\nu)}$$
$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$

The following geometrical design consideration(Yin et al., 2019) were assumed for concave bottom profile:

- $t_1 = (2.0 \sim 2.6)t = 2.3t$
- $t_2 = (1.8 \sim 2.2)t = 2t$
- $t_3 = (2.0 \sim 2.6)t = 2.3t$
- $r = (0.07 \sim 0.09)OD = 0.08OD$
- $h = (0.13 \sim 0.16)OD = 0.15OD$
- Transition$= 2h$

Where t is the thickness of the oxygen bottle wall and OD is the outer diameter of the oxygen bottle.

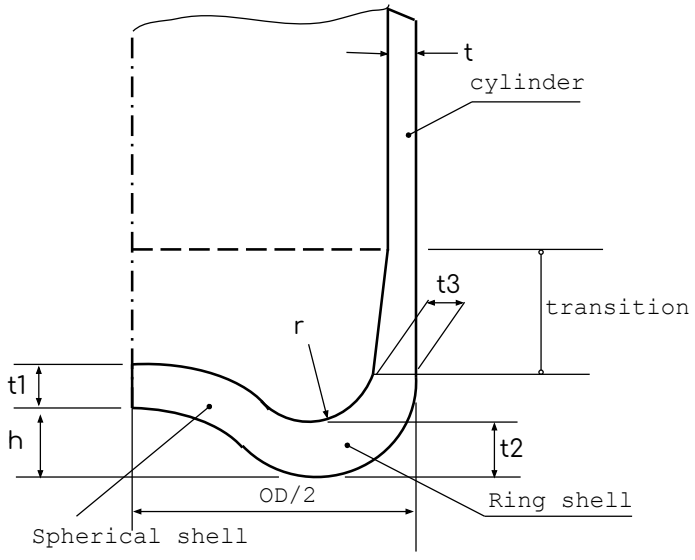


Figure 3.6: Cylinder Base Profile

The geometrical parameters of concave bottom profile, which was used for modelling are:

H	OD	t	t1	t2	t3	r	h	Transition
1365	232	5.4	12.42	11.88	12.42	18.56	33.64	67.28
		5.5	12.65	12.1	12.65	18.56	33.64	67.28
		5.6	12.88	12.32	12.88	18.56	33.64	67.28

Table 3.3: Concave Bottom Design Parameters

Note: All dimensions in table 3.3are in mm.

3.4 CAD Modelling

In order to incorporate the design of an oxygen cylinder into a simulation platform, computer-aided design (CAD) software was employed to create a highly accurate and detailed 3D model of the cylinder. This involved the inclusion of all relevant design components and specifications. A crucial consideration in this modeling process was determining the optimal wall thickness for the cylinder. To enable an analysis of the design during simulation, multiple versions of the model were generated with varying wall thicknesses. This facilitated an evaluation of the cylinder's strength and safety under different geometrical and material conditions and uses. Utilizing the SOLIDWORKS software, a precise and comprehensive model of the oxygen cylinder was produced. A snapshot of the resulting 3D model of the full-scale oxygen cylinder created using SOLIDWORKS is displayed below.

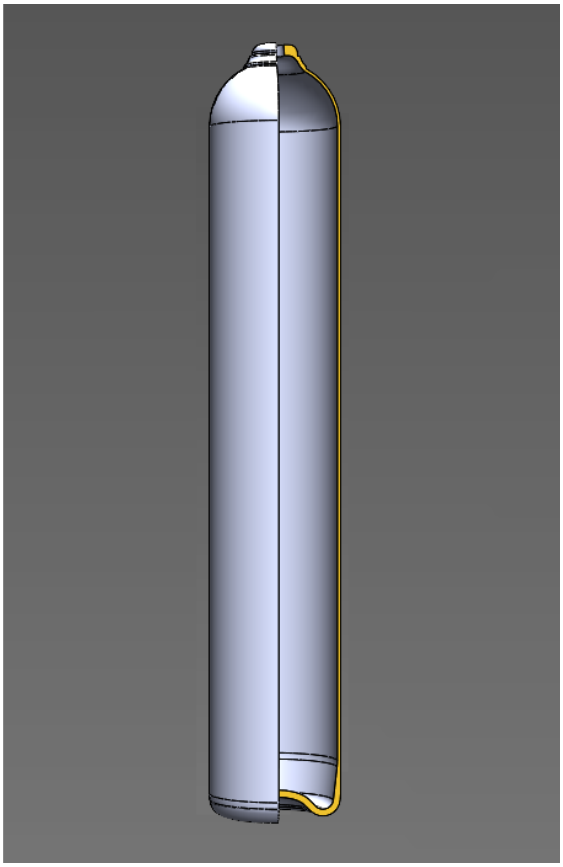


Figure 3.7: CAD Model in SOLIDWORKS

3.5 Simplification

In order to ensure an efficient and accurate simulation process for stress and deformation analysis, the original CAD model was simplified while maintaining the overall dimensions. This simplification involved the creation of a computationally streamlined version of the model that still effectively captured the essential design features and characteristics of the cylinder. The simplification of the CAD model increased the ability to reduce its complexity and streamline the simulation process, leading to the obtainment of approximated results for stress and deformation with greater efficiency and accuracy. The geometry of the simulation was generated in Gmsh, utilizing a combination of CAD models and manual input. The geometry was then partitioned into distinct regions to define the meshing areas. The meshing process involved the definition of the element size, selection of the meshing algorithm, and specification of the boundary conditions.

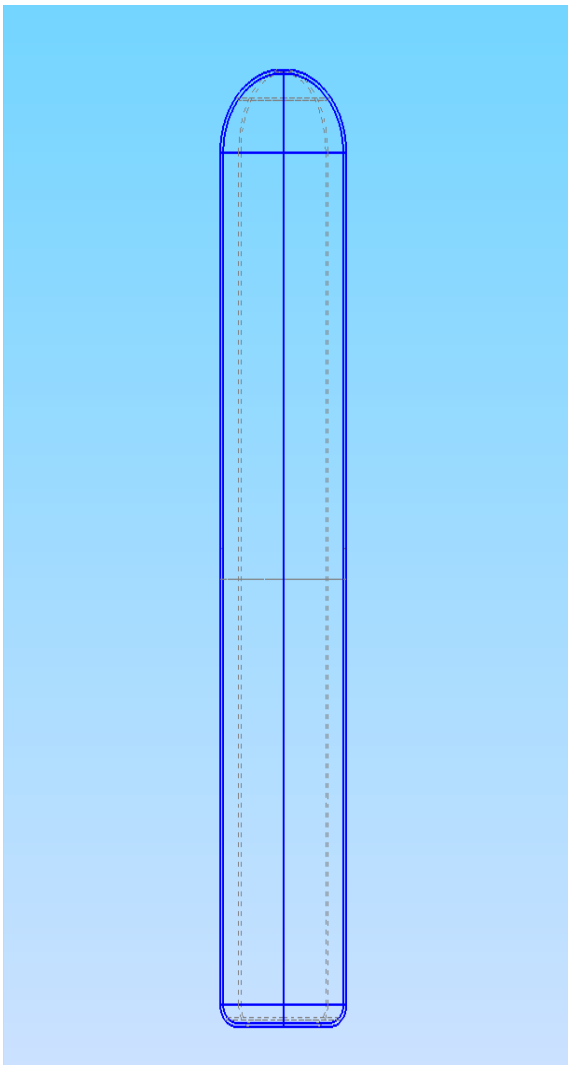


Figure 3.8: Simplified geometry in Gmsh.

3.6 FEA in ANSYS

FEA Simulations were also performed using ANSYS software to verify the program based on deal.ii since it is well-established and reliable FEA software. It enabled the cross-validation of the accuracy and reliability of the results, and to ensure that the findings were not specific to any particular software package. The same geometry and boundary conditions were applied and the corresponding hoop stress and deformations were noted for every model for each material.

3.7 Theoretical Calculation

Theoretical calculations were done to determine the hoop stress in the midsection of the cylinder using the theory from solid mechanics as mentioned in the theoretical background section. Hoop stress for each model of varying thickness was calculated and later compared with the FEA results. It may be noted that throughout the report, theoretical calculation and analytical calculation were used interchangeably.

3.8 FEA using Deal.ii

To conduct FEA in deal.ii, following steps were incorporated:

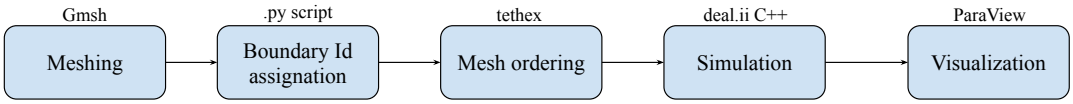


Figure 3.9: Algorithm for FEA in deal.ii

3.8.1 Meshing

The meshing stage of our project involved the process of discretizing the CAD model of the D-Type Medical oxygen cylinder into a finite number elements.In this study, Gmsh software was used to generate the mesh for the simulation. Gmsh is an open-source 3-D finite element grid generator with a build-in CAD engine and post-processor (Geuzaine & Remacle, 2009). It supports a variety of mesh types, including 1D, 2D, and 3D meshes, as well as structured and unstructured meshes. The *.geo* script used to create and mesh the geometry is provided in APPENDIX C.

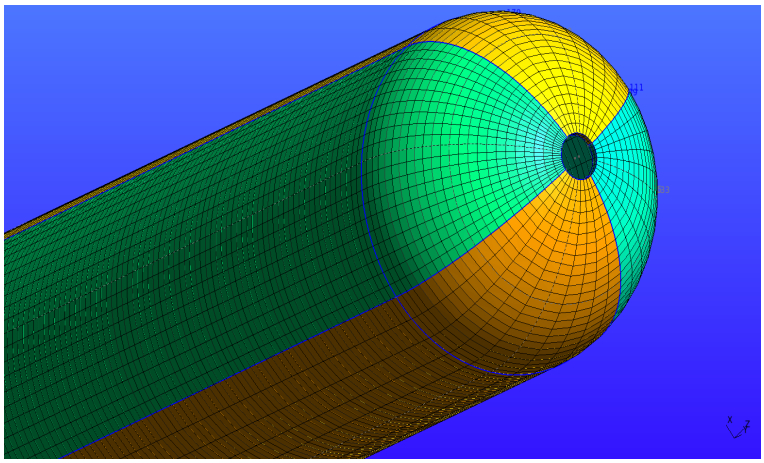


Figure 3.10: Simplified geometry after meshing in Gmsh.

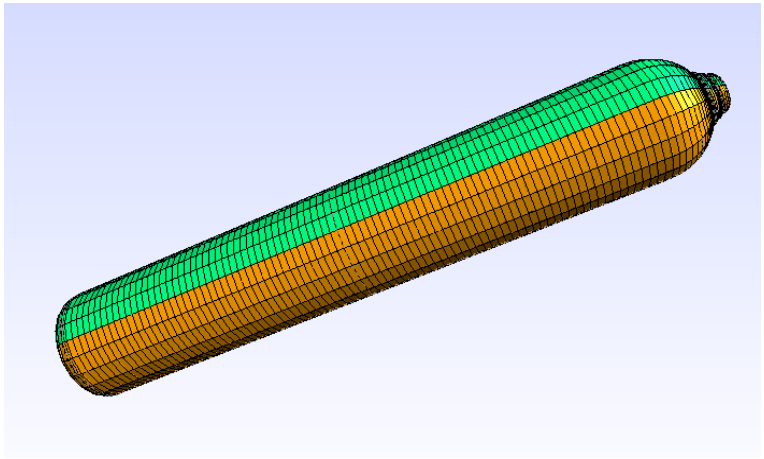


Figure 3.11: Actual geometry after meshing in Gmsh.

3.8.2 Boundary Id Assignation

The boundary id section involved assigning the appropriate boundary id to the solid model, to apply pressure and fixity constraints to the model during subsequent stages of the study. Boundary ids were to be applied in the form of physical ids in Gmsh. But the assignment of physical ids to the model in itself was not successful in this study as it assigned only geometric ids to the geometrical entities. So, the physical ids needed to be assigned manually. The meshed model being collection of large number of nodes and elements, it was not possible to do manually. So, a custom python script was made which assigns element with physical id which is unique for each geometric ids. Later on, Boundary conditions were applied specifically selecting those physical id's then taken as boundary id. The python script used in this step is provided in APPENDIX D.

3.8.3 Mesh Ordering

During the meshing process,the generated mesh was unstructured mesh of smaller elements that accurately represented the behavior of the geometry under different loads and conditions. However, to effectively perform the calculations using C++ code, it was necessary to rearrange these elements in a specific order generally called structured mesh. So, a C++ script, tethex (martemyev, 2013) was used to rearrange the mesh and convert tetrahedral elements to hexahedral elements that allowed our C++ code to efficiently perform the necessary calculations for stress and deformation analysis.

3.8.4 Simulation

To obtain displacement and stress, simulations of each CAD model of the Medical oxygen cylinder was run on C++ code, which was based on the deal.ii library. For simple geometries, simple program utilizing the shared memory system was successful. However, for actual cylinder CAD model, this program failed to converge due to the lack of enough memory. To this end,the distributed memory system using MPI into the program was incorporated. Nevertheless, stress calculation part could not be included into this code. Thus, only displacement in case of real cylinder was computed. The simulation part was done for three levels of geometry: Actual cylinder, Simplified cylinder and Hollow cylinder. For hollow cylinder (only cylindrical section), number of elements being comparatively small, both stresses and displacements were calculated. But for simplified and actual cylinder only displacements were calculated. We simulated our code for following material properties:

- 34Mn2V ($\lambda = 106.3\ GPa,\ \mu = 71.154\ GPa$)
- 34CrMo$_4$ ($\lambda = 105.44\ GPa,\ \mu = 76.356\ GPa$)
- 37MnSi5 ($\lambda = 118.84\ GPa,\ \mu = 79\ GPa$)

The boundary conditions applied in the simulation model included a fixed bottom surface and a pressure of 150 bar on the internal surfaces. The fixed bottom surface was implemented to prevent any movement or displacement of the system, while the 150 bar (15MPa) pressure on the internal surfaces provided a realistic representation of the operating conditions.

In this way, a number of simulations were conducted in our program. For deflection, C++ codes incorporating both shared memory system and distributed memory were able to be developed. However, for stress calculation, only shared memory system was used on a simple code. These simulations allowed to accurately model the behavior of the cylinder under different loads and conditions, and provided detailed information on its structural integrity and performance.The code developed for simulation are provided in APPENDIX B. First section of appendix gives code for stress and deformation calculation using shared

memory only and second section of appendix is about calculation of deformation by MPI.

3.8.5 Visualization

In order to visualize the simulation results, an open-source data visualization software, ParaView was used

3.9 Parametrization

The geometry of model was varied by adjusting the thickness of the cylinder, as well as testing the impact of different material properties on maximum hoop stress and maximum deformation. By simulating the model with these varying parameters, ability to better understanding of how changes in geometry and materials impacted the overall performance and safety of the Medical oxygen cylinder. These simulations allowed to optimize the design of the cylinder and ensure that it met all necessary performance and safety standards.

3.10 Verification and Validation

In the verification and validation section, results from the program simulation and theoretical calculations were compared and contrasted. In order to compare the results, the deformation and stress in the mid-section of the cylinder were taken into consideration. To further ensure the correctness of program, ANSYS simulation was conducted side-by-side. After a number of iterations and update, the results and the program were verified and validated.

3.11 Conclusion

After the careful examination of simulation results, conclusions were drawn about the strength and safety of the MOC for different materials and thicknesses. With this, documentation of the project was proceeded.

CHAPTER FOUR: RESULTS AND DISCUSSION

4.1 Hollow Cylinder

Hollow cylinder of different thickness i.e.,5.4, 5.5 and 5.6mm are assigned different materials and simulated in code as well as in ANSYS Mechanical,and following results were obtained. Further error on Hoop stress was calculated for deal.ii simulation with comparison to analytical calculation.

		Theoritical	Ansys		Deal.ii		% Error
Thickness	Material	Hoop Stress (MPa)	Hoop Stress (MPa)	Max Disp. (mm)	Hoop Stress (MPa)	Max Disp. (mm)	Hoop Stress
5.4	34Mn2V	322.2222	326.2764	0.6049	320.01	0.604	0.6859
5.4	34CrMo4	322.2222	325.6583	0.5511	320.15	0.550	0.6431
5.4	37MnSi5	322.2222	325.4323	0.5496	320.15	0.544	0.6440
5.5	34Mn2V	316.3636	318.4060	0.5932	313.90	0.592	0.7787
5.5	34CrMo4	316.3636	318.7604	0.5403	314.03	0.539	0.7376
5.5	37MnSi5	316.3636	318.3780	0.5388	314.03	0.534	0.7364
5.6	34Mn2V	310.7143	313.2900	0.5817	308.00	0.581	0.8749
5.6	34CrMo4	310.7143	313.3700	0.5299	308.11	0.529	0.8382
5.6	37MnSi5	310.7143	313.2500	0.2837	308.13	0.523	0.8327

Table 4.1: Comparison of simulation results for Hollow Cylinder

From the above table, error in Hoop stress is calculated by deal.ii simulation is under one percent (average error is 0.75 %) in comparison with theoretical calculation. Also the results from the deal.ii program has a close match with the result obtained with ANSYS simulations. The analysis revealed that our calculations were within a small margin of error compared to the commercial software, indicating the reliability of our methodology. This comparison of results of deal.ii based simulation to analytical calculation and ANSYS further provided a rigid belief that the algorithm used in this project work produces accurate results. It is also expected that the code works for all similar kind of geometrical interfaces as well as the code can be manipulated to work for different geometries.

Also. the deal.ii based code provided can be accessed and simulated for thin cylindrical geometries for various materials to understand further about the influence of materials to the deformation and stress so that a step toward optimization of material can be initiated. To better understand the relationship between material as well as geometric properties, it was plotted in figures 4.1 and 4.2.

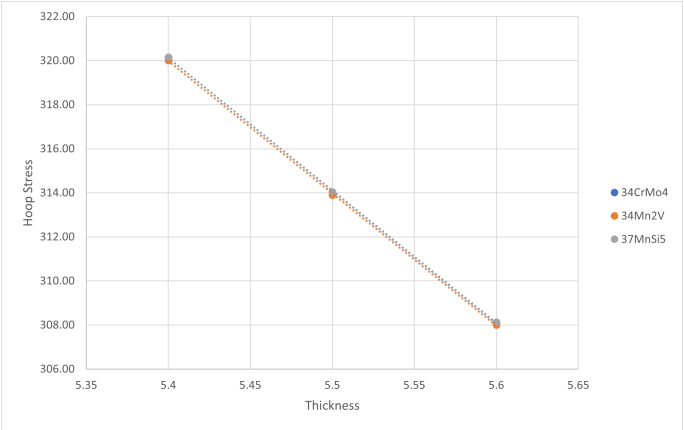


Figure 4.1: Hoop stress vs thickness for hollow cylinder

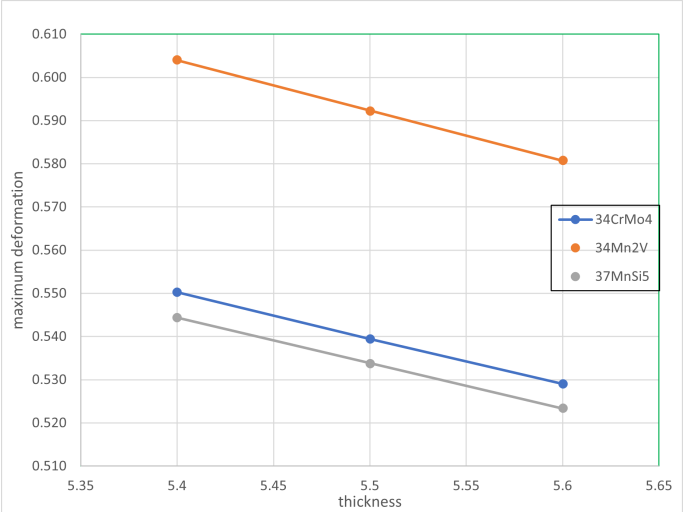


Figure 4.2: Deformation vs thickness for hollow cylinder

From the plot of thickness and hoop stress, it is clear that the stress is independent of choice of material and decreases with increase in thickness which perfectly aligned with theory that the stresses in thin wall cylinder is irrespective of material but is inversely proportional to thickness of section. Similarly, when the deformation vs thickness plot is analyzed it was found that the deformation also decreases with increase in thickness of the wall. When both of plots are generalized explicitly, a idea of relationship between hoop stress and deformation can be pictured mentally. Simply the relation of stress seems linearly proportional with deformation, this fact can be used when we have deformation to compare but no stresses values are provided. From the deformation vs thickness plot 4.2, the material 34Mn2V suffer from largest deformation whilst 37MnSi5 undergoes through the smallest deformation for the same thickness, and loading conditions, so the material 37MnSi5 could be more reliable to use for production of thin wall cylinders whose functionality are to provide a high pressure resistant work.

4.2 Simplified Cylinder

For the computational ease, simplified oxygen cylinder was simulated under different material and geometrical properties. The comparison of maximum deformations obtained from deal.ii and ANSYS simulation is shown in table 4.2 below:

Thickness	Material	deal.ii	ANSYS	% difference
		Maximum Disp.	Maximum Disp.	
5.4	34Mn2V	0.5415	0.5641	4.007
5.4	34CrMo4	0.5276	0.5488	3.868
5.4	37MnSi5	0.4869	0.4996	2.542
5.5	34Mn2V	0.5282	0.5184	1.890
5.5	34CrMo4	0.5147	0.5053	1.855
5.5	37MnSi5	0.4750	0.4589	3.518
5.6	34Mn2V	0.5152	0.5362	3.907
5.6	34CrMo4	0.5022	0.5219	3.775
5.6	37MnSi5	0.4633	0.4749	2.429

Table 4.2: Comparison of max. deformation (in mm) for simplified cylinder

It is to be noted that the % difference is obtained from dividing the difference by the ANSYS result in the table 4.2. It is evident that the results by open source simulation and ANSYS simulation are in close proximity. Moreover, they also agree on the distribution of deformation and location of maximum deformation on cylinder body which is inferred from the figures 4.3 and 4.4. Maximum deformation appears on the region of top head near the neck section. Thus, it can be concluded that the consideration of neck and head region is very crucial in designing a oxygen or pressurized gas cylinder.

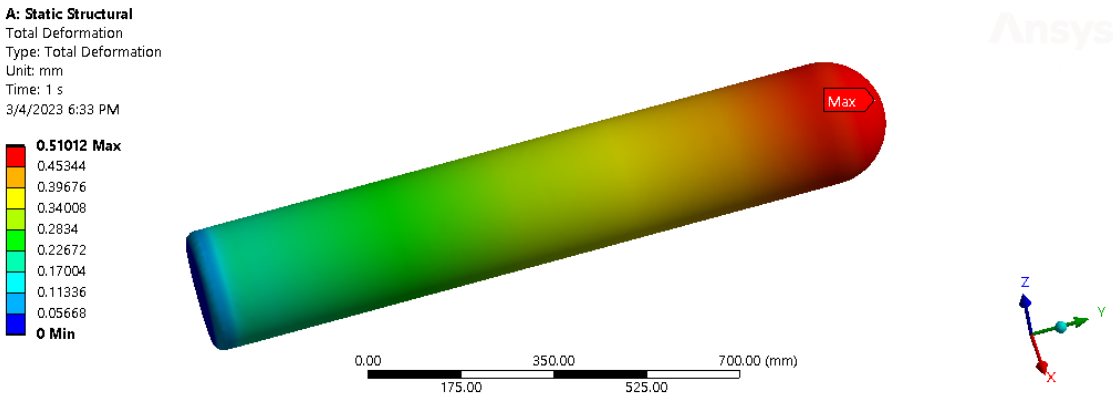


Figure 4.3: Displacement result from ANSYS(34Mn2V, 5.5mm)

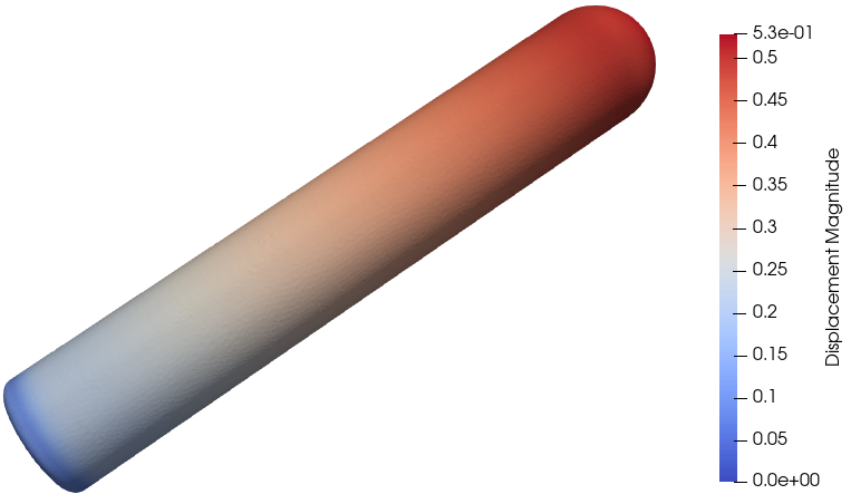


Figure 4.4: Displacement result from deal.ii (34Mn2V, 5.5mm)

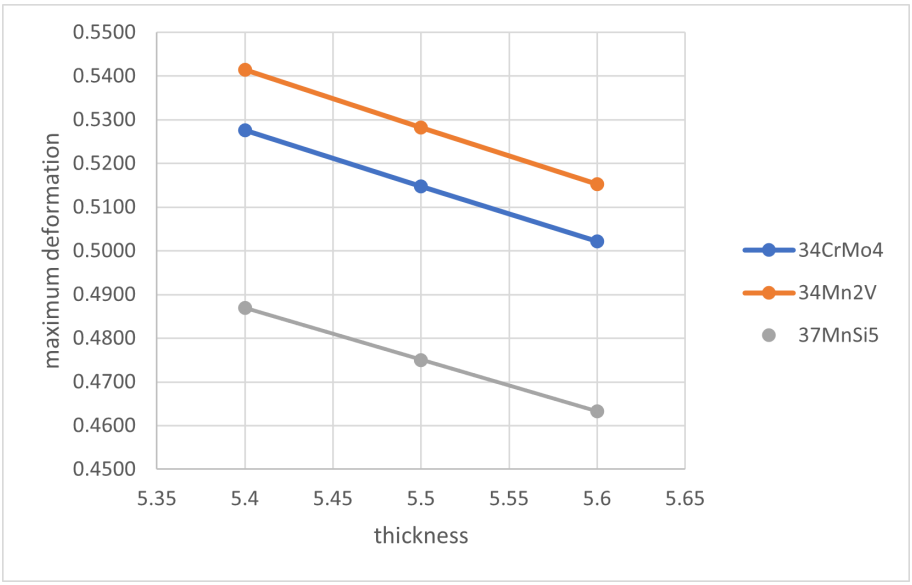


Figure 4.5: Deformation vs thickness for simplified cylinder

The maximum deformation vs thickness plot for different materials as shown in figure 4.5 suggest that the maximum deformation decreases for increasing thickness for all materials, also the correlation between maximum deformation and thickness seems linear as in the case of hollow cylinder. In sum up, it is found that cylinder made up of material 34Mn04 with thickness 5.6 mm is most safe with least deformation while that of 37MnSi5 with thickness 5.4 mm is relatively least safe.

4.3 Actual Cylinder

Actual model of oxygen cylinder was simulated under different material conditions for a nominal thickness of 6mm.The comparison for maximum deformation obtained from deal.ii and ANSYS Simulation is shown in Table below:

Material	ANSYS	deal.ii	% difference
34Mn2V	0.42601	0.44395	4.211
34CrMo4	0.41752	0.43151	3.351
37MnSi5	0.37625	0.39868	5.961

Table 4.3: Comparison of deformation (in mm) of actual cylinder

From the table, it is again found that the maximum deformation by deal.ii simulation and ANSYS simulation are quiet similar(within 6% difference range) as well as also the distribution of deformation and location of maximum deformation on cylinder body are similar for both which are shown in figures 4.6 and 4.7.

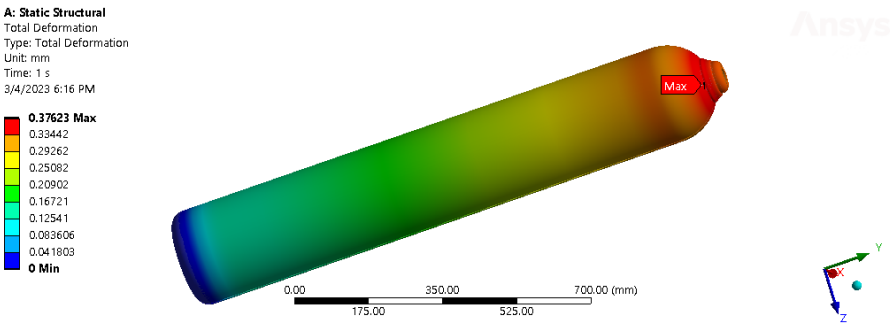


Figure 4.6: Displacement result from ANSYS (37MnSi5)

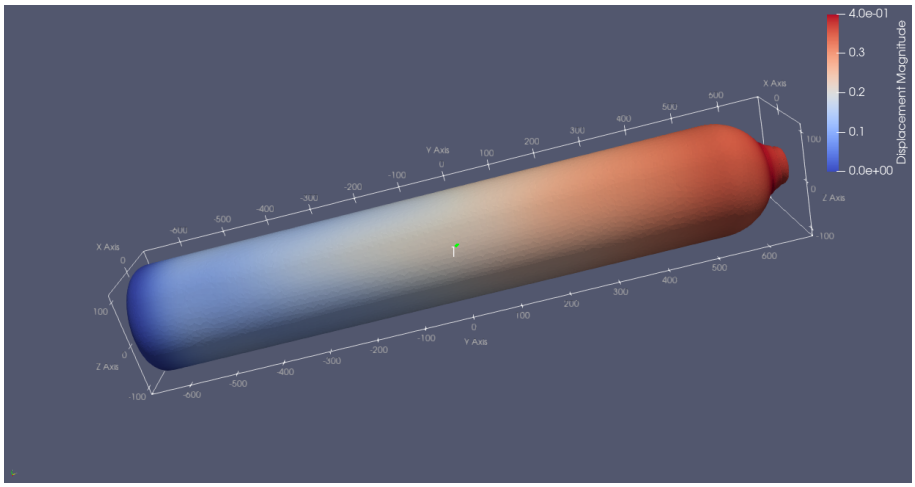


Figure 4.7: Displacement result from deal.ii (37MnSi5)

The results for other materials are placed in APPENDIX F. In all of the results the maximum deformation obtained was about 0.5mm which is very insignificant as compared to overall body dimension. This indicated that MOC is safe in 150 bar pressure . Nevertheless, in order to predict further about failure or safety we need to analyse the maximum and Von-misses stresses. But for actual real model, stress could not be calculated so, this remains as a part of shortcoming of this project. This was due to limit of computational power and time available. It also hampered the computation for actual oxygen cylinder model for different thicknesses. So, simulation was done for only one thickness for deformation whilst assigning different material properties.

Although the stress was not calculated in actual MOC model and simplified Model, it was done for hollow cylinder model which is a section of oxygen cylinder itself i.e., dimensions of cylinder are similar and just head and bottom profile are removed. From the comparison of simulation results in case of hollow cylinder, the program was verified. It was evident that the deformation was quiet similar for simplified and hollow cylinder and a little bit less in actual MOC. Also, hoop stress for all of them were found to be almost identical whilst the axial stress is very less in hollow cylinder as compared with that of actual and simplified MOC. In sum up, hoop stress was in the range of 300-350 Mpa and the axial stress was in the range of 130-180 Mpa manifesting that for thin-wall cylinder axial stress is half of the hoop stress. From these analyses, it was concluded that none of the geometry suffer maximum stress greater than 400 Mpa. Since, all the materials have Yield Stress of about 450 Mpa, this assured the safety of the vessel. From the results, it was inferred that MOCs are safe under working pressure of 150 bar.

CHAPTER FIVE: CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusion

In conclusion, this project successfully demonstrated the use of a deal.ii based program for analyzing stress and deformation on an oxygen cylinder. The methodology involved verification of the program by comparing the results with that of ANSYS and theoretical calculations in case of simple hollow cylinder in which maximum error was found to be 0.875 %. After that, the program was used to simulate the actual cylinder in which maximum deformation in the cylinder was found to be 0.44395 mm in case of 34Mn2V material with thickness 5.6 mm. For different materials, the simulation was repeated.

The findings of this project have important implications for the field of engineering and materials science, particularly for the design of safer and more efficient oxygen cylinders. The ability to accurately predict the stress and deformation on the cylinder can help engineers better understand the behavior of these cylinders under different conditions, such as changes in pressure or temperature, and design them accordingly. Additionally, the use of a deal.ii-based code for such simulations can be a valuable tool for researchers and engineers in various industries.

However, it is important to note that this project is not without its limitations. The simulations were conducted under certain assumptions and simplifications, and it is possible that more complex real-world scenarios may yield different results. Furthermore, the use of different simulation tools and software may also yield different results, and further studies can be conducted to compare the performance of different simulation tools for this application.

In conclusion, this project successfully demonstrated the use of a deal.ii-based code for analyzing stress and deformation on an oxygen cylinder. The results showed that the code was a reliable tool for simulating stress and deformation on the cylinder, and the findings have important implications for the field of engineering and materials science. Future studies can build upon these findings and further investigate the performance of different simulation tools and software for this application.

5.2 Recommendation

From the perspective of entire project work , concluding remarks and literature review, some recommendations were put forward to researchers,students or those who are interested in Finite analysis ,Oxygen Cylinder,Open Source Implementation.These recommendation can encourage readers to further investigate and build upon the findings of this study, ultimately contributing to a better understanding of the behavior of oxygen cylin-

ders and their optimization for use in various industries. The recommendations are:

1. Use of alternative simulation tools:
   While the deal.ii-based code was found to be a reliable tool for simulating stress and deformation on an oxygen cylinder, it may be worthwhile to investigate the use of other simulation tools and software for this application. This can help validate the results obtained from the current simulation and provide a more comprehensive understanding of the behavior of oxygen cylinders under different conditions.

2. Conducting experimental studies:
   While simulations can provide valuable insights into the behavior of oxygen cylinders, they may not always accurately represent real-world scenarios. Therefore, it may be worthwhile to conduct experimental studies to validate the findings obtained from the simulations and provide more accurate data for future analysis.

3. Optimization of cylinder design:
   The insights obtained from this study can be used to optimize the design of oxygen cylinders, making them safer and more efficient. Future studies can focus on incorporating different design features to enhance the performance of these cylinders under different conditions.

4. Investigation of other cylinder materials:
   This study focused on analyzing stress and deformation on an oxygen cylinder made from a specific set materials. However, it may be worthwhile to investigate the behavior of cylinders made from other materials like aluminium alloy , as the findings may differ depending on the material properties.

5. Further investigation into the effect of environmental conditions:
   The simulations conducted in this study assumed certain environmental conditions (Standard normal temperature and pressure). Future studies can investigate the effect of changes in environmental conditions such as temperature, humidity, and pressure on the stress and deformation of oxygen cylinders.

REFERENCES

Abdussalam. (2006). Finite element analysis of the design and manufacture of thin-walled pressure vessels used as aerosol cans.

Arndt, D., Bangerth, W., Davydov, D., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Pelteret, J.-P., Turcksin, B., & Wells, D. (2021a). The deal.ii finite element library: Design, features, and insights [Development and Application of Open-source Software for Problems with Numerical PDEs]. *Computers Mathematics with Applications*, *81*, 407–422. https://doi.org/https://doi.org/10.1016/j.camwa.2020.02.022

Arndt, D., Bangerth, W., Davydov, D., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Pelteret, J.-P., Turcksin, B., & Wells, D. (2021b). The deal.II finite element library: Design, features, and insights. *Computers &amp Mathematics with Applications*, *81*, 407–422. https://doi.org/10.1016/j.camwa.2020.02.022

Arndt, D., Bangerth, W., Heltai, K., Maier, M., pelteret, T., & Wells, O. (2022). *The deal.ii finite element library*. https://www.dealii.org (accessed: 01.04.2022)

Bangerth, W., Hartmann, R., & Kanschat, G. (2007). Deal.ii—a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, *33*, 24.

Bultel, H., & Vogt, J.-B. (2010). Influence of heat treatment on fatigue behaviour of 4130 aisi steel [Fatigue 2010]. *Procedia Engineering*, *2*(1), 917–924. https://doi.org/https://doi.org/10.1016/j.proeng.2010.03.099

Chaves, E. W. (2013). *Notes on continuum mechanics* (1st ed.). International Center for Numerical Methods in Engineering (CIMNE).

Cinatl, E. (2018). Finite element discretizations for linear elasticity. *All Theses*. https://tigerprints.clemson.edu/all_theses/2977

Das, P., & Islam, M. S. (2019). Structural analysis of a thick-walled pressure vessel using fem. *Journal of Engineering Science*, *10*(2), 69–78. https://www2.kuet.ac.bd/JES/

Geuzaine, C., & Remacle, J.-F. (2009). Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, *79*(11), 1309–1331. https://doi.org/https://doi.org/10.1002/nme.2579

Ioannis Koutromanos, J. M., & Roy, C. (2018). *Fundamentals of finite element analysis: Linear finite element analysis* (1st ed.). Wiley.

Kronbichler, M., & Kormann, K. (2012). A generic interface for parallel cell-based finite element operator application. *Computers & Fluids*, *63*, 135–147.

Li, Y., Fang, W., Lu, C., Gao, Z., Ma, X., Jin, W., Ye, Y., & Wang, F. (2019). Microstructure and mechanical properties of 34crmo4 steel for gas cylinders formed

by hot drawing and flow forming. *Materials*, *12*, 1351. https://doi.org/10.3390/ma12081351

Macwan, S. P., Hu, Z., & Delfanian, F. (2011). Experimental verification of model pressurized thick-walled cylinder with numerical and theoretical methods. *Volume 8: Mechanics of Solids, Structures and Fluids; Vibration, Acoustics and Wave Propagation*, 173–178. https://doi.org/10.1115/IMECE2011-65763

martemyev. (2013). Tethex [January, 2023]. https://github.com/martemyev/tethex

Mirzaei, M. (2008). Failure analysis of an exploded gas cylinder. *Enegineering Failure Analysis*, *15*(7), 289–305. https://doi.org/https://doi.org/10.1016/j.engfailanal.2007.11.005

Mohamed, A. (2018). Finite element analysis for stresses in thin-walled pressurized steel cylinders.

Nendra Wibawa, L. A., Diharjo, K., Raharjo, W. W., & H. Jihad, B. (2021). Stress analysis of thick-walled cylinder for rocket motor case under internal pressure. *Journal of Advanced Research in Fluid Mechanics and Thermal Sciences*, *70*(2), 106–115. https://akademiabaru.com/submit/index.php/arfmts/article/view/2958

P. Palanivelu, R. S. P. (2017). A paper on design and analysis of pressure vessel. *International Journal of Engineering Research & Technology (IJERT)*, *06*. http://dx.doi.org/10.17577/IJERTV6IS060424

Rangari, L. D. (2012). Finite element analysis of lpg gas cylinder. *International Journal of Applied Research in Mechanical Engineering*, *2*. https://doi.org/10.47893/IJARME.2012.1055

Wang, H., Yao, X., Li, L., Sang, Z., & Krakauer, B. W. (2017). Full length article. *Thin-Walled Structures*, *113*(100), 104–110. https://doi.org/10.1016/j.tws.2017.01.018

Wang, H., Zheng, T., Sang, Z., & Krakauer, B. (2021). Burst pressures of thin-walled cylinders constructed of steel exhibiting a yield plateau. *International Journal of Pressure Vessels and Piping*, *193*, 104483. https://doi.org/https://doi.org/10.1016/j.ijpvp.2021.104483

Yin, Z., Su, T., & He, M. (2019). Gas packaging container based on ansys finite element analysis and structural optimization design. *Journal of Physics: Conference Series*, *1187*, 032089. https://doi.org/10.1088/1742-6596/1187/3/032089

APPENDIX A: MATHEMATICAL DERIVATION

Strong Form

Suppose we have a three-dimensional solid elastic body. The body is subjected to body forces $\vec{b}$ acting in the domain denoted by Ω and traction $\vec{t}$ on the boundary denoted by Γ_t with displacements $\vec{d}$ assigned on the boundary denoted by Γ_d as shown in figure A.1 below.

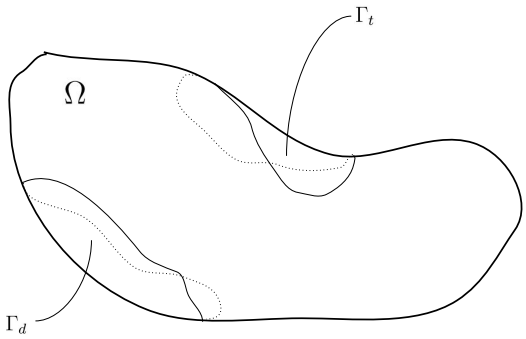


Figure A.1: Solid Body

To derive the governing equation, we consider a small cuboid element inside the domain. The stresses and body forces on the cuboid are acting as shown in figure A.2.

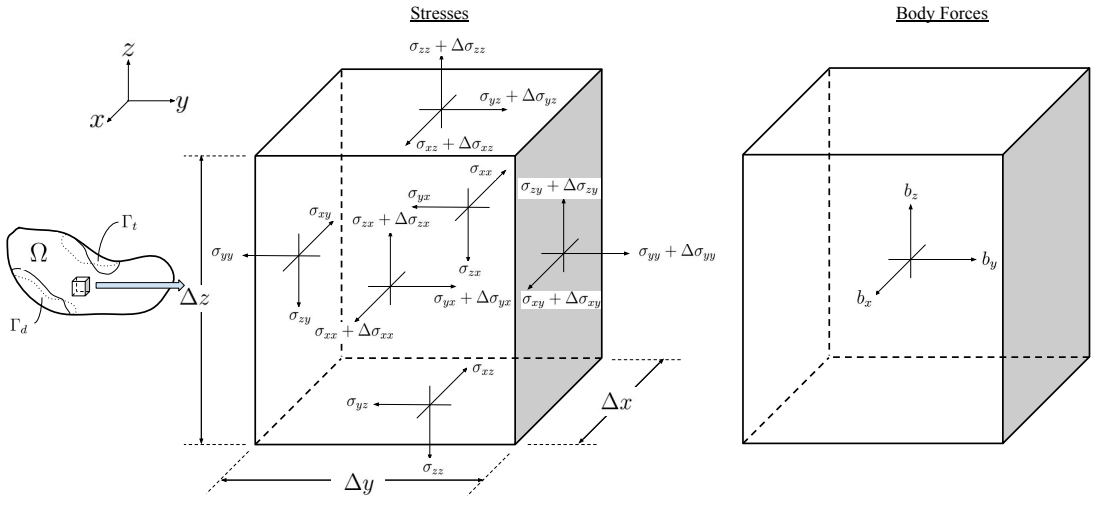


Figure A.2: Cuboid element

We can obtain three equilibrium equations for the element,

$$\begin{aligned}
\{(\sigma_{xx}+\Delta\sigma_{xx})-\sigma_{xx}\}\cdot\Delta y\Delta z+\{(\sigma_{xy}+\Delta\sigma_{xy})-\sigma_{xy}\}\cdot\Delta x\Delta z+\\
\{(\sigma_{xz}+\Delta\sigma_{xz})-\sigma_{xz}\}\cdot\Delta x\Delta y+b_x\cdot\Delta x\Delta y\Delta z=0\\
\{(\sigma_{yx}+\Delta\sigma_{yx})-\sigma_{yx}\}\cdot\Delta y\Delta z+\{(\sigma_{yy}+\Delta\sigma_{yy})-\sigma_{yy}\}\cdot\Delta x\Delta z+\\
\{(\sigma_{yz}+\Delta\sigma_{yz})-\sigma_{yz}\}\cdot\Delta x\Delta y+b_y\cdot\Delta x\Delta y\Delta z=0\\
\{(\sigma_{zx}+\Delta\sigma_{zx})-\sigma_{zx}\}\cdot\Delta y\Delta z+\{(\sigma_{zy}+\Delta\sigma_{zy})-\sigma_{zy}\}\cdot\Delta x\Delta z+\\
\{(\sigma_{zz}+\Delta\sigma_{zz})-\sigma_{zz}\}\cdot\Delta x\Delta y+b_z\cdot\Delta x\Delta y\Delta z=0
\end{aligned}$$

Expanding and then dividing each equation by the volume of the element, i.e. $\Delta x\Delta y\Delta z$,

$$\begin{aligned}
\frac{\Delta\sigma_{xx}}{\Delta x}+\frac{\Delta\sigma_{xy}}{\Delta y}+\frac{\Delta\sigma_{xz}}{\Delta z}+b_x=0\\
\frac{\Delta\sigma_{yx}}{\Delta x}+\frac{\Delta\sigma_{yy}}{\Delta y}+\frac{\Delta\sigma_{yz}}{\Delta z}+b_y=0\\
\frac{\Delta\sigma_{zx}}{\Delta x}+\frac{\Delta\sigma_{zy}}{\Delta y}+\frac{\Delta\sigma_{zz}}{\Delta z}+b_z=0
\end{aligned}$$

Now, if we take the limit i.e. $\lim_{\Delta x\to 0}$, $\lim_{\Delta y\to 0}$, and $\lim_{\Delta z\to 0}$, we get,

$$\begin{aligned}
\frac{\partial\sigma_{xx}}{\partial x}+\frac{\partial\sigma_{xy}}{\partial y}+\frac{\partial\sigma_{xz}}{\partial z}+b_x=0\\
\frac{\partial\sigma_{yx}}{\partial x}+\frac{\partial\sigma_{yy}}{\partial y}+\frac{\partial\sigma_{yz}}{\partial z}+b_y=0\\
\frac{\partial\sigma_{zx}}{\partial x}+\frac{\partial\sigma_{zy}}{\partial y}+\frac{\partial\sigma_{zz}}{\partial z}+b_z=0
\end{aligned}$$

In vector notation,

$$\begin{aligned}
\vec{\nabla}\cdot\vec{\sigma}_x+b_x=0\\
\vec{\nabla}\cdot\vec{\sigma}_y+b_y=0\\
\vec{\nabla}\cdot\vec{\sigma}_z+b_z=0
\end{aligned}$$

which can be written as,

$$\text{div}(\boldsymbol{\sigma}(\vec{u}))+\vec{b}=0$$

This is the governing equation for the domain. We need to get the governing equation for entire closed domain, i.e. $\bar{\Omega}=\Omega\cup\Gamma$.

For the boundary with traction, consider a tetrahedral element with the diagonal face aligning with the boundary surface Γ_t. The stresses and prescribed traction are acting on the element as shown in figure A.3. Before proceeding with derivation, we consider the vector

surface on which traction acts, is $\Delta\vec{S}$ with direction cosines n_x, n_y, and n_z such that,

$$\Delta S_x = \Delta S \cdot n_x$$
$$\Delta S_y = \Delta S \cdot n_y$$
$$\Delta S_z = \Delta S \cdot n_z, \text{where, } \Delta S = |\Delta\vec{S}|$$

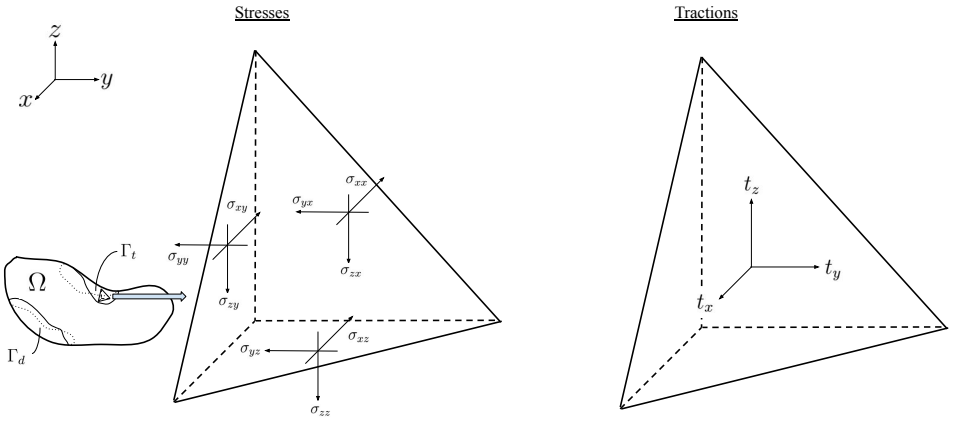


Figure A.3: Tetrahedral element

Now, for tetrahedral element, three equilibrium equations are as follows,

$$-\sigma_{xx} \cdot \Delta S_x - \sigma_{xy} \cdot \Delta S_y - \sigma_{xz} \cdot \Delta S_z + t_x \cdot \Delta S = 0$$
$$-\sigma_{yx} \cdot \Delta S_x - \sigma_{yy} \cdot \Delta S_y - \sigma_{yz} \cdot \Delta S_z + t_y \cdot \Delta S = 0$$
$$-\sigma_{zx} \cdot \Delta S_x - \sigma_{zy} \cdot \Delta S_y - \sigma_{zz} \cdot \Delta S_z + t_z \cdot \Delta S = 0$$

Dividing each term by ΔS, we get,

$$-\sigma_{xx} \cdot \frac{\Delta S_x}{\Delta S} - \sigma_{xy} \cdot \frac{\Delta S_y}{\Delta S} - \sigma_{xz} \cdot \frac{\Delta S_z}{\Delta S} + t_x \cdot = 0$$
$$-\sigma_{yx} \cdot \frac{\Delta S_x}{\Delta S} - \sigma_{yy} \cdot \frac{\Delta S_y}{\Delta S} - \sigma_{yz} \cdot \frac{\Delta S_z}{\Delta S} + t_y \cdot = 0$$
$$-\sigma_{zx} \cdot \frac{\Delta S_x}{\Delta S} - \sigma_{zy} \cdot \frac{\Delta S_y}{\Delta S} - \sigma_{zz} \cdot \frac{\Delta S_z}{\Delta S} + t_z \cdot = 0$$

Substituting the expressions of direction cosines, we get,

$$-\sigma_{xx} \cdot n_x - \sigma_{xy} \cdot n_y - \sigma_{xz} \cdot n_z + t_x = 0$$
$$-\sigma_{yx} \cdot n_x - \sigma_{yy} \cdot n_y - \sigma_{yz} \cdot n_z + t_y = 0$$
$$-\sigma_{zx} \cdot n_x - \sigma_{zy} \cdot n_y - \sigma_{zz} \cdot n_z + t_z = 0$$

Now, rearranging and writing each equation in vector notation,

$$\vec{\sigma}_x \cdot \vec{n} = t_x$$
$$\vec{\sigma}_y \cdot \vec{n} = t_y$$
$$\vec{\sigma}_z \cdot \vec{n} = t_z$$

In more compact form,

$$\boldsymbol{\sigma}(\vec{u}) \cdot \vec{n} = \vec{t}$$

Finally, for boundary Γ_d, where displacements are specified, following relation can be written,

$$u_x = d_x$$
$$u_y = d_y$$
$$u_z = d_z$$

In vector notation,

$$\vec{u} = \vec{d}$$

Finally, the strong form for the small displacement three-dimensional linear elasticity problem with Neumann and Dirichlet boundary conditions is,
Given $\vec{b} : \Omega \to \mathbb{R}^3$, $\vec{d} : \Gamma_d \to \mathbb{R}^3$, $\vec{t} : \Gamma_t \to \mathbb{R}^3$, find $\vec{u} : \bar{\Omega} \to \mathbb{R}^3$ such that,

$$\begin{aligned} -\text{div}(\boldsymbol{\sigma}(\vec{u})) &= \vec{b} \text{ in } \Omega \\ \vec{u} &= \vec{d} \text{ on } \Gamma_d \\ \boldsymbol{\sigma}(\vec{u}) \cdot \vec{n} &= \vec{t} \text{ on } \Gamma_t \end{aligned}$$

Weak Form

In order to be able to work with Finite Element Method, we need to obtain the weak form of the governing equations derived. We take the vector product of strong form with the test function vector $\vec{v}$ belonging to the Hilbert Space **H** and integrate the equation for entire domain, we get,

$$
\begin{aligned}
-\int_\Omega \text{div}(\boldsymbol{\sigma}(\vec{u})) \cdot \vec{v} &= \int_\Omega \vec{b} \cdot \vec{v} \\
\int_\Omega \boldsymbol{\sigma}(\vec{u}) : \nabla \vec{v} - \int_\Gamma (\boldsymbol{\sigma}(\vec{u}) \cdot \vec{n}) \cdot \vec{v} &= \int_\Omega \vec{b} \cdot \vec{v} \\
\int_\Omega \boldsymbol{\sigma}(\vec{u}) : \nabla \vec{v} - \int_{\Gamma_d} (\boldsymbol{\sigma}(\vec{u}) \cdot \vec{n}) \cdot \vec{v} - \int_{\Gamma_t} (\boldsymbol{\sigma}(\vec{u}) \cdot \vec{n}) \cdot \vec{v} &= \int_\Omega \vec{b} \cdot \vec{v} \\
\int_\Omega \boldsymbol{\sigma}(\vec{u}) : \nabla \vec{v} - \int_{\Gamma_t} \vec{t} \cdot \vec{v} &= \int_\Omega \vec{b} \cdot \vec{v} \\
\int_\Omega \boldsymbol{\sigma}(\vec{u}) : \nabla \vec{v} &= \int_{\Gamma_t} \vec{t} \cdot \vec{v} + \int_\Omega \vec{b} \cdot \vec{v}
\end{aligned}
$$

In bi-linear form,

$$a(\vec{u}, \vec{v}) = (\vec{t}, \vec{v})_{\Gamma_t} + (\vec{b}, \vec{v})$$

In this way, weak form of the problem is,
Given $\vec{b} : \Omega \to \mathbb{R}^3,\ \vec{d} : \Gamma_d \to \mathbb{R}^3,\ \vec{t} : \Gamma_t \to \mathbb{R}^3$, find $\vec{u} : \bar{\Omega} \to \mathbb{R}^3$ such that for $\vec{v} \in \mathbf{H}$,

$$a(\vec{u}, \vec{v}) \quad = \quad (\vec{t}, \vec{v})_{\Gamma_t} + (\vec{b}, \vec{v}) \text{ in } \Omega$$

Note:

$$
\begin{aligned}
a(\vec{u}, \vec{v}) &= \int_\Omega \boldsymbol{\sigma}(\vec{u}) : \nabla \vec{v} \\
&= \int_\Omega \boldsymbol{\sigma}(\vec{u}) : \varepsilon(\vec{v}) \\
&= \int_\Omega (\lambda \, (\text{div } \vec{u}) \, \boldsymbol{I} + 2\mu\varepsilon(\vec{u})) : \varepsilon(\vec{v}) \\
&= \int_\Omega \lambda \, (\text{div } \vec{u}) \, \boldsymbol{I} : \varepsilon(\vec{v}) + 2\mu\varepsilon(\vec{u}) : \varepsilon(\vec{v}) \\
&= \int_\Omega \lambda \, (\text{div } \vec{u}) \, (\text{div } \vec{v}) + 2\mu\varepsilon(\vec{u}) : \varepsilon(\vec{v}) \qquad \text{(Cinatl, 2018)}
\end{aligned}
$$

Finite Element Approximation

Since, working with infinite-dimensional solution is difficult and not feasible and so on, we take the subset of the infinite-dimensional Hilbert space, i.e., we consider finite numbers of nodes. The domain is discretized into small finite numbers of elements. This discretized domain is called mesh. Thus, considering N nodes, vector-valued approximated solution and test function (global) are re-written as,

$$\vec{u} \approx \vec{u^h} = \left\{ \begin{array}{l} u_x^h = \sum_{i=0}^{N} U_{ix}\phi_i(\mathbf{x}) \\ u_y^h = \sum_{j=0}^{N} U_{jy}\phi_j(\mathbf{y}) \\ u_z^h = \sum_{k=0}^{N} U_{ky}\phi_k(\mathbf{z}) \end{array} \right\} \text{ and } \vec{v} \approx \vec{v^h} = \left\{ \begin{array}{l} v_x^h = \sum_{i=0}^{N} V_{ix}\phi_i(\mathbf{x}) \\ v_y^h = \sum_{j=0}^{N} V_{jy}\phi_j(\mathbf{y}) \\ v_z^h = \sum_{k=0}^{N} V_{kz}\phi_k(\mathbf{z}) \end{array} \right\}$$

In deal.ii, this is achieved in following manner,,
We express vector-valued basis function as,

$$\Phi_i(\mathbf{x}) = \phi_i(\mathbf{x})\ \mathbf{e}_{\text{comp}(i)}$$

Thus, vector-valued test function can be represented as,

$$\vec{v^h} = \begin{bmatrix} \phi_0 & 0 & 0 & \phi_3 & 0 & 0 & \phi_6 & 0 & \cdots & 0 \\ 0 & \phi_1 & 0 & 0 & \phi_4 & 0 & 0 & \phi_7 & \cdots & 0 \\ 0 & 0 & \phi_2 & 0 & 0 & \phi_5 & 0 & 0 & \cdots & \phi_{3N-1} \end{bmatrix} * \begin{Bmatrix} V_0 \\ V_1 \\ V_2 \\ \vdots \\ V_{3N-3} \\ V_{3N-2} \\ V_{3N-1} \end{Bmatrix}$$

Similarly, solution vector can be represented as,

$$\vec{u^h} = \begin{bmatrix} \phi_0 & 0 & 0 & \phi_3 & 0 & 0 & \phi_6 & 0 & \cdots & 0 \\ 0 & \phi_1 & 0 & 0 & \phi_4 & 0 & 0 & \phi_7 & \cdots & 0 \\ 0 & 0 & \phi_2 & 0 & 0 & \phi_5 & 0 & 0 & \cdots & \phi_{3N-1} \end{bmatrix} * \begin{Bmatrix} U_0 \\ U_1 \\ U_2 \\ \vdots \\ U_{3N-3} \\ U_{3N-2} \\ U_{3N-1} \end{Bmatrix}$$

where, N is the number of nodes. It is to be noted that each consecutive set of three values

in the coefficient column vector represents vector displacement at the corresponding node. For example, the set (U_0, U_1, U_2) represents vector displacement at node 0.
In sum up, the solution in deal.ii is approximated as ,

$$\vec{u}(\mathbf{x}) \approx \vec{u^h}(\mathbf{x}) = \sum_j \Phi_j(\mathbf{x})\, U_j$$

Also, the test function is approximated as,

$$\vec{v}(\mathbf{x}) \approx \vec{v^h}(\mathbf{x}) = \sum_i \Phi_i(\mathbf{x})\, V_i$$

Substituting these expressions, we get,

$$\begin{aligned}
a(\vec{u},\vec{v}) &\approx a(\vec{u^h},\vec{v^h}) \\
&= \int_\Omega \lambda \left(\text{div } \vec{u^h}\right)\left(\text{div } \vec{v^h}\right) + 2\mu\varepsilon(\vec{u^h}) : \varepsilon(\vec{v^h}) \\
&= \lambda \int_\Omega \text{div } \left(\sum_j \Phi_j U_j\right) \text{div } \left(\sum_i \Phi_i V_i\right) + 2\mu \int_\Omega \varepsilon\left(\sum_j \Phi_j U_j\right) \varepsilon\left(\sum_i \Phi_i V_i\right) \\
&= \sum_i \sum_j \left[\lambda \int_\Omega (\text{div } \Phi_i)(\text{div } \Phi_j) + 2\mu \int_\Omega \varepsilon\left(\Phi_i\right) : \varepsilon\left(\Phi_j\right)\right] U_j V_i
\end{aligned}$$

And,

$$\begin{aligned}
(\vec{t},\vec{v})_{\Gamma_t} &\approx (\vec{t},\vec{v^h})_{\Gamma_t} \\
&= \int_{\Gamma_t} \vec{t} \cdot \sum_i \Phi_i V_i \\
&= \sum_i \left[\int_{\Gamma_t} \vec{t} \cdot \Phi_i\right] V_i
\end{aligned}$$

Also,

$$\begin{aligned}
(\vec{b},\vec{v}) &\approx (\vec{b},\vec{v^h}) \\
&= \int_\Omega \vec{b} \cdot \sum_i \Phi_i V_i \\
&= \sum_i \left[\int_\Omega \vec{b} \cdot \Phi_i\right] V_i
\end{aligned}$$

Finally, the approximated weak form becomes,

$$a(\vec{u^h}, \vec{v^h}) = (\vec{t}, \vec{v^h})_{\Gamma_t} + (\vec{b}, \vec{v^h})$$

$$\sum_i \sum_j \left[\lambda \int_\Omega (\text{div } \Phi_i)(\text{div } \Phi_j) + 2\mu \int_\Omega \varepsilon(\Phi_i) : \varepsilon(\Phi_j)\right] U_j V_i = \sum_i \left[\int_{\Gamma_t} \vec{t} \cdot \Phi_i\right] V_i + \sum_i \left[\int_\Omega \vec{b} \cdot \Phi_i\right] V_i$$

$$\sum_i \left[\sum_j \left[\lambda \int_\Omega (\text{div } \Phi_i)(\text{div } \Phi_j) + 2\mu \int_\Omega \varepsilon(\Phi_i) : \varepsilon(\Phi_j)\right] U_j\right] V_i = \sum_i \left[\left[\int_{\Gamma_t} \vec{t} \cdot \Phi_i\right] + \left[\int_\Omega \vec{b} \cdot \Phi_i\right]\right] V_i$$

$$\sum_i \left[\sum_j \left[\lambda \int_\Omega (\text{div } \Phi_i)(\text{div } \Phi_j) + 2\mu \int_\Omega \varepsilon(\Phi_i) : \varepsilon(\Phi_j)\right] U_j = \left[\int_{\Gamma_t} \vec{t} \cdot \Phi_i\right] + \left[\int_\Omega \vec{b} \cdot \Phi_i\right]\right] V_i$$

By definition, V_i cannot be zero for all j from 0 to 3N-1, thus column vector $\{V\}$ cannot be zero. Therefore, for $i = 0$ to $3N - 1$, the expression inside the big brackets must be valid, i.e.,
for $i = 0$ to $3N - 1$,

$$\sum_j A_{ij} U_j = F_i$$

In matrix form ,

AU = F

Where,

$$\begin{aligned}
A_{ij} &= \lambda \int_\Omega (\text{div } \Phi_i)(\text{div } \Phi_j) + 2\mu \int_\Omega \varepsilon(\Phi_i) : \varepsilon(\Phi_j) \\
&= \lambda \sum_{k,l} (\partial_l(\Phi_i)_l, \partial_k(\Phi_j)_k)_\Omega + 2\mu * \left(\frac{1}{2}\left(\sum_{k,l} (\partial_k(\Phi_i)_l, \partial_k(\Phi_j)_l)_\Omega + \sum_{k,l} (\partial_k(\Phi_i)_l, \partial_l(\Phi_j)_k)_\Omega\right)\right) \\
&= \lambda \sum_{k,l} (\partial_l(\Phi_i)_l, \partial_k(\Phi_j)_k)_\Omega + \mu \left(\sum_{k,l} (\partial_k(\Phi_i)_l, \partial_k(\Phi_j)_l)_\Omega + \sum_{k,l} (\partial_k(\Phi_i)_l, \partial_l(\Phi_j)_k)_\Omega\right) \\
&= \sum_{k,l} (\lambda \partial_l(\Phi_i)_l, \partial_k(\Phi_j)_k)_\Omega + \sum_{k,l} (\mu \partial_k(\Phi_i)_l, \partial_k(\Phi_j)_l)_\Omega + \sum_{k,l} (\mu \partial_k(\Phi_i)_l, \partial_l(\Phi_j)_k)_\Omega \\
&= \sum_{k,l} \{(\lambda \partial_l(\Phi_i)_l, \partial_k(\Phi_j)_k)_\Omega + (\mu \partial_k(\Phi_i)_l, \partial_k(\Phi_j)_l)_\Omega + (\mu \partial_k(\Phi_i)_l, \partial_l(\Phi_j)_k)_\Omega\}
\end{aligned}$$

here, l and k run through 0 to dim-1.
Similarly,

$$\begin{aligned}
F_i &= \int_{\Gamma_t} \vec{t} \cdot \Phi_i + \int_\Omega \vec{b} \cdot \Phi_i \\
&= \sum_l (t_l, (\Phi_i)_l)_{\Gamma_t} + \sum_l (b_l, (\Phi_i)_l)_\Omega \\
&= \sum_l \{(t_l, (\Phi_i)_l)_{\Gamma_t} + (b_l, (\Phi_i)_l)_\Omega\}
\end{aligned}$$

Now, for each cell $K \in T \approx \Omega$ and the cell face $X \in T_b \approx \Gamma_t$, local cell matrix and right hand side vectors are given by,

$$\begin{aligned}
A_{ij}^K &= \sum_{k,l}\{(\lambda\partial_l(\Phi_i)_l, \partial_k(\Phi_j)_k)_K + (\mu\partial_k(\Phi_i)_l, \partial_k(\Phi_j)_l)_K + (\mu\partial_k(\Phi_i)_l, \partial_l(\Phi_j)_k)_K\} \\
&= \left(\lambda\partial_{\text{comp}(i)}\phi_i, \partial_{\text{comp}(j)}\phi_j\right)_K + \sum_l (\mu\partial_l\phi_i, \partial_l\phi_j)_K \ \delta_{\text{comp}(i),\text{comp}(j)} + \left(\mu\partial_{\text{comp}(j)}\phi_i, \partial_{\text{comp}(i)}\phi_j\right)_K \\
&= \left(\lambda\partial_{\text{comp}(i)}\phi_i, \partial_{\text{comp}(j)}\phi_j\right)_K + (\mu\nabla\phi_i, \nabla\phi_j)_K \ \delta_{\text{comp}(i),\text{comp}(j)} + \left(\mu\partial_{\text{comp}(j)}\phi_i, \partial_{\text{comp}(i)}\phi_j\right)_K \\
&= \left(\lambda\partial_{\text{comp}(i)}\phi_i, \partial_{\text{comp}(j)}\phi_j\right)_K + \left(\mu\partial_{\text{comp}(j)}\phi_i, \partial_{\text{comp}(i)}\phi_j\right)_K + (\mu\nabla\phi_i, \nabla\phi_j)_K \ \delta_{\text{comp}(i),\text{comp}(j)}
\end{aligned}$$

It should be noted that for local cell matrix, i and j run through 0 to $3 \cdot n - 1$ where n is the no. of nodes on the cell.
And,

$$\begin{aligned}
F_i^K &= \sum_l\{(t_l, (\Phi_i)_l)_X + (b_l, (\Phi_i)_l)_K\} \\
&= \sum_l (t_l, (\Phi_i)_l)_X + \sum_l \left(b_l, \phi_i\delta_{l,\text{comp}(i)}\right)_K \\
&= \left(t_{\text{comp}(i)}, \phi_i\right)_X + \left(b_{\text{comp}(i)}, \phi_i\right)_K
\end{aligned}$$

Quadrature

Since integration on real cell is difficult, we use mapping to compute the integration. In this, we map the real cell to the reference cell using **Jacobian**,
Reference (parent) cell is just tri-unit cube. Since, the integral limits are from -1 to 1, now we can use Gauss Quadrature formula to calculate the integration numerically.

$$\begin{aligned}
\left(\lambda\partial_{\text{comp}(i)}\phi_i, \partial_{\text{comp}(j)}\phi_j\right)_K &\approx \sum_q \lambda\left(\partial_{\text{comp}(i)}\phi_i(q)\right)\left(\partial_{\text{comp}(j)}\phi_j(q)\right)\text{JxW}(q) \\
\mu\left(\partial_{\text{comp}(j)}\phi_i, \partial_{\text{comp}(i)}\phi_j\right)_K &\approx \sum_q \mu\left(\partial_{\text{comp}(j)}\phi_i(q)\right)\left(\partial_{\text{comp}(i)}\phi_j(q)\right)\text{JxW}(q) \\
(\mu\nabla\phi_i, \nabla\phi_j)_K \ \delta_{\text{comp}(i),\text{comp}(j)} &\approx \sum_q \mu\left(\nabla\phi_i(q)\nabla\phi_j(q)\right)_K \delta_{\text{comp}(i),\text{comp}(j)}\text{JxW}(q) \\
\left(t_{\text{comp}(i)}, \phi_i\right)_X &\approx \sum_{q_f}\left(t_{\text{comp}(i)}\phi_i(q_f)\right)\text{JxW}(qf) \\
\left(b_{\text{comp}(i)}, \phi_i\right)_K &\approx \sum_q \left(b_{\text{comp}(i)}\phi_i(q)\right)\text{JxW}(q)
\end{aligned}$$

where, JxW(q) is the product of determinant of **J**acobian and the weights corresponding to the quadrature point q.

In the problem, traction is that of pressure, i.e.,

$$\vec{t} = -P \cdot \vec{n}$$

where, P is the pressure acting on the surface and $\vec{n}$ is the normal surface vector of the surface. Thus, traction is the pressure acting on the surface normally.
Then,

$$\sum_{q_f} \left(t_{\text{comp}(i)} \phi_i(q_f)\right) \text{JxW}(q) \quad = \quad \sum_{q_f} \left(-P \cdot n_{\text{comp}(i)} \phi_i(q_f)\right) \text{JxW}(q)$$

Substituting the expressions,

$$A_{ij}^K = \sum_q \{\lambda \left(\partial_{\text{comp}(i)} \phi_i(q)\right) \left(\partial_{\text{comp}(j)} \phi_j(q)\right) + \mu \left(\partial_{\text{comp}(j)} \phi_i(q)\right) \left(\partial_{\text{comp}(i)} \phi_j(q)\right) + \mu \left(\nabla \phi_i(q)\right) \left(\nabla \phi_j(q)\right) \delta_{\text{comp}(i),\text{comp}(j)}\} \text{JxW}(q)$$

Also,

$$F_i^K = \sum_{q_f} \left(-P \cdot n_{\text{comp}(i)} \phi_i(q_f)\right) \text{JxW}(q_f) + \sum_q \left(b_{\text{comp}(i)} \phi_i(q)\right) \text{JxW}(q)$$

where q_f refers to the quadrature points on the face of the cell.

Stress Calculation

The constitutive relation between the stress and strain is,

$$\boldsymbol{\sigma}(\vec{u}) = \mathcal{C} : \varepsilon(\vec{u})$$

where, $\mathcal{C}$ is rank-4 coefficient(stress-strain) tensor, and $\varepsilon(\vec{u})$ is strain.
For isotropic material, elements of coefficient tensor are given by,

$$c_{ijkl} = \lambda \delta_{ij} \delta_{kl} + \mu \left(\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}\right)$$

where, δ_{ij}'s are Kronecker delta functions. And the strain is given by,

$$\epsilon(\vec{u})_{kl} = \frac{1}{2}(\partial_k \vec{u}_l + \partial_l \vec{u}_k)$$

In tensorial notation:

$$\varepsilon(\vec{u}) = \frac{1}{2}\left(\nabla \vec{u} + (\nabla \vec{u})^T\right)$$

Stress transformation

Since, the formulation is worked on cartesian coordinate system, we need to transform the stress into polar coordinate system to obtain hoop and radial stresses.
For this, a transformation matrix is constructed,

$$\mathbf{P} = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where, $\theta = \arctan \frac{y}{x}$.
Therefore, if the stress in cartesian coordinate system is,

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

and stress in polar coordinate system is,

$$\boldsymbol{\sigma}_P = \begin{bmatrix} \sigma_{rr} & \sigma_{r\theta} & \sigma_{rz} \\ \sigma_{\theta r} & \sigma_{\theta\theta} & \sigma_{\theta z} \\ \sigma_{zr} & \sigma_{z\theta} & \sigma_{zz} \end{bmatrix}$$

then, the transformation relation is given by,

$$\boldsymbol{\sigma}_P = [\mathbf{P}][\boldsymbol{\sigma}_C][\mathbf{P}]^T$$

Accessing the diagonal elements of $\boldsymbol{\sigma}_P$ gives the radial stress, hoop stress and axial stress respectively.

APPENDIX B: CODE FOR SIMULATION

Code For Deformation and Stress

```
#include <deal.II/base/symmetric_tensor.h>
#include <deal.II/base/tensor.h>
#include<deal.II/base/timer.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/base/tensor.h>
#include<deal.II/physics/transformations.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_q.h>
#include <fstream>
#include <iostream>
#include <vector>
#include <algorithm>

namespace Program
{
  using namespace dealii;

    template <int dim>
  SymmetricTensor<4, dim> get_stress_strain_tensor(const double lambda,
                                                   const double mu)
  {
    SymmetricTensor<4, dim> tmp;
    for (unsigned int i = 0; i < dim; ++i)
      for (unsigned int j = 0; j < dim; ++j)
```

```
      for (unsigned int k = 0; k < dim; ++k)
        for (unsigned int l = 0; l < dim; ++l)
          tmp[i][j][k][l] = (((i == k) && (j == l) ? mu : 0.0) +
                             ((i == l) && (j == k) ? mu : 0.0) +
                             ((i == j) && (k == l) ? lambda : 0.0));
  return tmp;
}

template <int dim>
inline SymmetricTensor<2, dim>
get_strain(const std::vector<Tensor<1, dim>> &grad)
{
  Assert(grad.size() == dim, ExcInternalError());

  SymmetricTensor<2, dim> strain;
  for (unsigned int i = 0; i < dim; ++i)
    strain[i][i] = grad[i][i];

  for (unsigned int i = 0; i < dim; ++i)
    for (unsigned int j = i + 1; j < dim; ++j)
      strain[i][j] = (grad[i][j] + grad[j][i]) / 2;

  return strain;
}
// to transform stress from cartesian to polar coordinate system
template <int dim>
inline SymmetricTensor<2,dim>
cart_to_polar(const Point<dim> &point,
                        const SymmetricTensor<2,dim> &stressC)
{
  const double theta = atan(point[1]/point[0]);
  const double sintheta = sin(theta);
  const double costheta = cos(theta);

  Tensor<2,dim> lambda({{ costheta, sintheta, 0},
                        { -sintheta, costheta, 0},
                        { 0, 0, 1}});
  SymmetricTensor<2,dim> stressP;
  for (unsigned int i = 0; i < 3; ++i) {
         for (unsigned int k = 0; k < 3; ++k) {
           double tmp2=0;
           for (unsigned int j = 0; j < 3; ++j) {
             double tmp = 0;
             for (unsigned int l = 0; l <3; ++l)
               tmp += stressC[j][l]*lambda[k][l];
             tmp2 += lambda[i][j]*tmp;
           }
```

```
90              stressP[i][k] = tmp2;
91            }
92  }
93    return stressP;
94  }
95
96  template <int dim>
97  class ElasticProblem
98  {
99  public:
100   ElasticProblem();
101   void run();
102
103 private:
104   void setup_system();
105   void assemble_system();
106   void solve();
107   void stressCalc();
108   void refine_grid();
109   void output_results(const unsigned int cycle) const;
110
111   Triangulation<dim> triangulation;
112   DoFHandler<dim> dof_handler;
113
114   FESystem<dim> fe;
115
116   AffineConstraints<double> constraints;
117
118   SparsityPattern sparsity_pattern;
119   SparseMatrix<double> system_matrix;
120
121   Vector<double> solution;
122   Vector<double> system_rhs;
123   std::vector<SymmetricTensor<2,dim>> stress;
124   Vector<double> norm_of_stress;
125
126   Vector<double> radial_stress;
127   Vector<double> hoop_stress;
128   Vector<double> axial_stress;
129
130   Vector<double> residual;
131   static const SymmetricTensor<4, dim> stress_strain_tensor;
132   const QGauss<dim> quadrature_formula;
133 };
134
135 template <int dim>
136 const SymmetricTensor<4, dim> ElasticProblem<dim>::stress_strain_tensor =
```

```
137   get_stress_strain_tensor<dim>(/*lambda = */ 12.232e10,
138                                 /*mu = */ 7.e10);
139 template <int dim>
140 class BOdyForceValues : public Function<dim>
141 {
142 public:
143   BOdyForceValues();
144
145   virtual void vector_value(const Point<dim> &p,
146                             Vector<double> & values) const override;
147
148   virtual void
149   vector_value_list(const std::vector<Point<dim>> &points,
150                     std::vector<Vector<double>> & value_list) const override;
151 };
152
153
154 template <int dim>
155 BOdyForceValues<dim>::BOdyForceValues()
156   : Function<dim>(dim)
157 {}
158
159
160 template <int dim>
161 inline void BOdyForceValues<dim>::vector_value(const Point<dim> & /*p*/,
162                                        Vector<double> &values) const
163 {
164   Assert(values.size() == dim, ExcDimensionMismatch(values.size(), dim));
165
166   const double g = 9.81;
167   const double rho = 7850;
168
169   values = 0;
170   values(dim - 1) = -rho * g;
171 }
172
173
174
175 template <int dim>
176 void BOdyForceValues<dim>::vector_value_list(
177   const std::vector<Point<dim>> &points,
178   std::vector<Vector<double>> & value_list) const
179 {
180   const unsigned int n_points = points.size();
181
182   Assert(value_list.size() == n_points,
183          ExcDimensionMismatch(value_list.size(), n_points));
```

```
184
185     for (unsigned int p = 0; p < n_points; ++p)
186       BOdyForceValues<dim>::vector_value(points[p], value_list[p]);
187   }
188
189   template<int dim>
190   class PressureBoundaryValues : public Function<dim>
191   {
192     public:
193     PressureBoundaryValues(): Function<dim>(1)
194     {}
195
196     virtual double value(const Point<dim> & p,const unsigned int component = 0)
197                                                       const override;
198
199   };
200
201   template <int dim>
202   double PressureBoundaryValues<dim>::value(const Point<dim> & /*p*/,
203                                 const unsigned int /*component*/) const
204   {
205     return -1.5e07;
206   }
207
208
209   template <int dim>
210   ElasticProblem<dim>::ElasticProblem()
211     : dof_handler(triangulation)
212     , fe(FE_Q<dim>(1), dim),quadrature_formula(fe.degree + 1)
213   {}
214
215
216
217   template <int dim>
218   void ElasticProblem<dim>::setup_system()
219   {
220     dof_handler.distribute_dofs(fe);
221     solution.reinit(dof_handler.n_dofs());
222     system_rhs.reinit(dof_handler.n_dofs());
223
224     constraints.clear();
225     DoFTools::make_hanging_node_constraints(dof_handler, constraints);
226     VectorTools::interpolate_boundary_values(dof_handler,
227                                              0,
228                                              Functions::ZeroFunction<dim>(dim),
229                                              constraints);
230     constraints.close();
```

```
231
232     DynamicSparsityPattern dsp(dof_handler.n_dofs(), dof_handler.n_dofs());
233     DoFTools::make_sparsity_pattern(dof_handler,
234                                     dsp,
235                                     constraints,
236                                     /*keep_constrained_dofs = */ false);
237     sparsity_pattern.copy_from(dsp);
238
239     system_matrix.reinit(sparsity_pattern);
240   }
241
242
243
244   template <int dim>
245   void ElasticProblem<dim>::assemble_system()
246   {
247     QGauss<dim> quadrature_formula(fe.degree + 1);
248
249     QGauss<dim-1> face_quadrature_formula(fe.degree +1);
250
251     FEFaceValues<dim> fe_face_values(fe,
252                           face_quadrature_formula,
253                           update_values | update_normal_vectors|
254                           update_quadrature_points | update_JxW_values);
255
256     const unsigned int n_face_q_points = face_quadrature_formula.size();
257
258     FEValues<dim> fe_values(fe,
259                           quadrature_formula,
260                           update_values | update_gradients |
261                           update_quadrature_points | update_JxW_values);
262
263     const unsigned int dofs_per_cell = fe.n_dofs_per_cell();
264     const unsigned int n_q_points = quadrature_formula.size();
265
266     FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
267     Vector<double> cell_rhs(dofs_per_cell);
268
269     std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
270
271     std::vector<double> lambda_values(n_q_points);
272     std::vector<double> mu_values(n_q_points);
273     std::vector<double> pressure_values(n_face_q_points);
274
275     Functions::ConstantFunction<dim> lambda(12.232e10), mu(7.9e10);
276
277     BOdyForceValues<dim> gravity;
```

```
278     PressureBoundaryValues<dim> pressure_boundary;
279
280     std::vector<Vector<double>> rhs_values(n_q_points,Vector<double>(dim));
281
282     for (const auto &cell : dof_handler.active_cell_iterators())
283       {
284         cell_matrix = 0;
285         cell_rhs = 0;
286
287         fe_values.reinit(cell);
288
289         lambda.value_list(fe_values.get_quadrature_points(), lambda_values);
290         mu.value_list(fe_values.get_quadrature_points(), mu_values);
291
292         gravity.vector_value_list(fe_values.get_quadrature_points(), rhs_values);
293
294         for (const unsigned int i : fe_values.dof_indices())
295           {
296             const unsigned int component_i =
297               fe.system_to_component_index(i).first;
298
299         for (const unsigned int j : fe_values.dof_indices())
300               {
301                 const unsigned int component_j =
302                   fe.system_to_component_index(j).first;
303
304          for (const unsigned int q_point :
305                       fe_values.quadrature_point_indices())
306             {
307             cell_matrix(i, j) +=
308                 (
309                 (fe_values.shape_grad(i, q_point)[component_i] * //d_comp(i)(phi_i(
                        ↪ q))
310                 fe_values.shape_grad(j, q_point)[component_j] * //d_comp(j)(phi_j(q
                        ↪ ))
311                 lambda_values[q_point]) //lambda
312                 +
313                 (fe_values.shape_grad(i, q_point)[component_j] * //d_comp(i)(phi_j(
                        ↪ q))
314                 fe_values.shape_grad(j, q_point)[component_i] * //d_comp(j)(phi_i(q
                        ↪ ))
315                 mu_values[q_point]) //mu
316                 + //
317                 ((component_i == component_j) ? //for Kronecker delta
318                 (fe_values.shape_grad(i, q_point) * //grad(phi_i(q))
319                 fe_values.shape_grad(j, q_point) * //grad(phi_i(q))
320                 mu_values[q_point]) : //mu
```

```
321                 0)
322                        ) *
323                 fe_values.JxW(q_point); //JxW(q)
324                   }
325               }
326           }
327
328         for (const unsigned int i : fe_values.dof_indices())
329           {
330             const unsigned int component_i =
331               fe.system_to_component_index(i).first;
332
333             for (const unsigned int q_point :
334                 fe_values.quadrature_point_indices())
335               cell_rhs(i) += rhs_values[q_point][component_i]* //b_comp(i)
336                              fe_values.shape_value(i, q_point)* //phi_i(q)
337                              fe_values.JxW(q_point); //JxW(q)
338           }
339
340       for(unsigned int face_number =0;
341                          face_number < GeometryInfo<dim> :: faces_per_cell;
342                         ++face_number)
343           {
344       if((cell->face(face_number)->at_boundary()) &&
345                        (cell->face(face_number)->boundary_id()== 2))
346             {
347               fe_face_values.reinit(cell,face_number);
348               pressure_boundary.value_list(fe_face_values.get_quadrature_points(),
349                                                           pressure_values);
350
351           for(unsigned int i=0; i<dofs_per_cell; ++i)
352             {
353              const unsigned int component_i =
354                   fe.system_to_component_index(i).first;
355           for(const unsigned int q_index : fe_face_values.quadrature_point_indices
                ↪ ())
356             {
357            cell_rhs(i) += (pressure_values[q_index] *
358                        fe_face_values.normal_vector(q_index)[component_i]) * //p*
                             ↪ vec(n)
359                        fe_face_values.shape_value(i,q_index) * //phi_i(q)
360                        fe_face_values.JxW(q_index); //JxW(q)
361                 }
362               }
363             }
364           }
365
```

```
366         cell->get_dof_indices(local_dof_indices);
367         constraints.distribute_local_to_global(
368           cell_matrix, cell_rhs, local_dof_indices, system_matrix, system_rhs);
369       }
370   }
371
372
373
374
375   template <int dim>
376   void ElasticProblem<dim>::solve()
377   {
378     SolverControl solver_control(3e3, 1e-2);
379     SolverCG<Vector<double>> cg(solver_control);
380
381     PreconditionJacobi<SparseMatrix<double>> preconditioner;
382     preconditioner.initialize(system_matrix);
383
384     cg.solve(system_matrix, solution, system_rhs, preconditioner);
385     stressCalc();
386
387     residual.reinit(dof_handler.n_dofs());
388     system_matrix.vmult(residual, solution);
389     residual -= system_rhs;
390     std::cout << "Iterations required : "
391               << solver_control.last_step() << '\n'
392               << "Max norm of residual: "
393               << residual.linfty_norm() << '\n';
394     constraints.distribute(solution);
395   }
396
397  template<int dim>
398  void ElasticProblem<dim>::stressCalc()
399  {
400   FEValues<dim> fe_values(fe,
401                           quadrature_formula,
402                           update_values|update_gradients);
403   std::vector<std::vector<Tensor<1,dim>>> solution_grads(
404                                     quadrature_formula.size(),
405                                     std::vector<Tensor<1,dim>>(dim));
406
407   radial_stress.reinit(triangulation.n_active_cells());
408   hoop_stress.reinit(triangulation.n_active_cells());
409   axial_stress.reinit(triangulation.n_active_cells());
410
411
```

```
412   std::vector<SymmetricTensor<2,dim>> stress_local(triangulation.n_active_cells()
          ↪ );
413   std::vector<SymmetricTensor<2,dim>> stress_local_polar(triangulation.
          ↪ n_active_cells());
414   Vector<double> norm_of_stress_local(triangulation.n_active_cells());
415   {
416     for(auto &cell : dof_handler.active_cell_iterators())
417     {
418       fe_values.reinit(cell);
419       fe_values.get_function_gradients(solution,solution_grads);
420
421       const unsigned int cell_index = cell->active_cell_index();
422       const Point<dim> cell_center = cell->center();
423       SymmetricTensor<2,dim> accumulated_stress ;
424       accumulated_stress = 0;
425       for(unsigned int q = 0; q < quadrature_formula.size(); ++q)
426       {
427         const SymmetricTensor<2,dim> quad_stress = (stress_strain_tensor *
428                                                     get_strain(solution_grads[q]));
429         accumulated_stress += quad_stress;
430       }
431       stress_local[cell_index] = accumulated_stress/quadrature_formula.size();
432       norm_of_stress_local(cell_index)= accumulated_stress.norm();
433
434       const SymmetricTensor<2,dim> tmp = stress_local[cell_index];
435
436       stress_local_polar[cell_index] = cart_to_polar<dim>(cell_center,tmp);
437
438       radial_stress[cell_index] = stress_local_polar[cell_index][0][0];
439       hoop_stress[cell_index] = stress_local_polar[cell_index][1][1];
440       axial_stress[cell_index] = stress_local_polar[cell_index][2][2];
441     }
442   }
443   stress = stress_local;
444   norm_of_stress = norm_of_stress_local;
445 }
446
447
448   template <int dim>
449   void ElasticProblem<dim>::refine_grid()
450   {
451     Vector<float> estimated_error_per_cell(triangulation.n_active_cells());
452
453     KellyErrorEstimator<dim>::estimate(dof_handler,
454                                        QGauss<dim - 1>(fe.degree + 1),
455                                        {},
456                                        solution,
```

```
                                    estimated_error_per_cell);

  GridRefinement::refine_and_coarsen_fixed_number(triangulation,
                                                  estimated_error_per_cell,
                                                  0.3,
                                                  0.03);

  triangulation.execute_coarsening_and_refinement();
}



template <int dim>
void ElasticProblem<dim>::output_results(const unsigned int cycle) const
{
  std::vector<std::string> solution_names(dim, "Displacement");

  std::vector<DataComponentInterpretation::DataComponentInterpretation>
    solution_component_interpretation(
      dim, DataComponentInterpretation::component_is_part_of_vector);
  DataOut<dim> data_out;
  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(solution,
                           solution_names,
                           DataOut<dim>::type_dof_data,
                           solution_component_interpretation);
  // printing component stresses
  data_out.add_data_vector(radial_stress,"radial_stress",DataOut<dim>::
      ↪ type_cell_data);
  data_out.add_data_vector(hoop_stress,"hoop_stress", DataOut<dim>::
      ↪ type_cell_data);
  data_out.add_data_vector(axial_stress,"axial_stress", DataOut<dim>::
      ↪ type_cell_data);
  //printing norm of stresses
  data_out.add_data_vector(norm_of_stress,"norm_of_stress",DataOut<dim>::
      ↪ type_cell_data);

  data_out.build_patches();

  std::ofstream output("37MnSi5 5.4mm simple hollow-" + std::to_string(cycle) +
      ↪  ".vtk");
  data_out.write_vtk(output);
}
```

```
499  template <int dim>
500  void ElasticProblem<dim>::run()
501  {
502    Timer timer;
503    for (unsigned int cycle = 0; cycle < 1; ++cycle)
504      {
505        std::cout << "Cycle "<< cycle << ':' << std::endl;
506
507        if (cycle == 0)
508          {
509            // mesh generation for simple hollow cylinder
510            const double OD =232, THK = 5.4, Height = 1200, scale =1;
511            const double outer_radius = scale *OD/2,
512                         inner_radius = (outer_radius - scale * THK);
513            GridGenerator::cylinder_shell(triangulation,
514                                          scale*Height,
515                                          inner_radius,
516                                          outer_radius);
517            for(const auto &cell : triangulation.active_cell_iterators())
518            for (const auto &face : cell->face_iterators())
519            if(face->at_boundary())
520            {
521              const Point<dim> face_center = face->center();
522              if(std::fabs(face_center[2]) <1e-12)
523              face->set_boundary_id(0);
524              else if (std:: fabs(face_center[2] - (scale * Height)) <1e-12 )
525              face-> set_boundary_id(1);
526              else if (std::sqrt(face_center[0] * face_center[0] +
527                              face_center[1]*face_center[1]) <
528                              (inner_radius + outer_radius)/2)
529              face->set_boundary_id(2);
530              else
531              face->set_boundary_id(3);
532            }
533          }
534        else
535        {
536        std::cout << "Refining.... "<< std::endl;
537          refine_grid();
538        std::cout << "...complete! "<< std::endl;
539        }
540        std::cout<< "time Elapsed: "<< timer.cpu_time() << "sec." << std:: endl;
541
542        std::cout << "Number of active cells: "
543                  << triangulation.n_active_cells() << std::endl;
544
545        std::cout << "Setting up.... "<< std::endl;
```

```
546         setup_system();
547         std::cout << "...complete! "<< std::endl;
548         std::cout<< "time Elapsed: "<< timer.cpu_time() << "sec." << std:: endl;
549
550
551         std::cout << "Number of degrees of freedom: "<< dof_handler.n_dofs()
552                   << std::endl;
553         std::cout << "Assembling.... "<< std::endl;
554         assemble_system();
555         std::cout << "...complete! "<< std::endl;
556         std::cout<< "time Elapsed: "<< timer.cpu_time() << "sec." << std:: endl;
557
558         std::cout << "Solving.... "<< std::endl;
559         solve();
560         std::cout << "...complete! "<< std::endl;
561         std::cout<< "time Elapsed: "<< timer.cpu_time() << "sec." << std:: endl;
562
563         std::cout << "Outputting.... "<< std::endl;
564         output_results(cycle);
565         std::cout << "...complete! "<< std::endl;
566
567
568         std::cout<< "Total time: "<< timer.cpu_time() << "sec." << std:: endl;
569   }
570 }
571 } // namespace Program
572
573
574 int main()
575 {
576   try
577     {
578       Program::ElasticProblem<3> elastic_problem_3d;
579       elastic_problem_3d.run();
580     }
581   catch (std::exception &exc)
582     {
583       std::cerr << std::endl
584                 << std::endl
585                 << "----------------------------------------------------"
586                 << std::endl;
587       std::cerr << "Exception on processing: "<< std::endl
588                 << exc.what() << std::endl
589                 << "Aborting!" << std::endl
590                 << "----------------------------------------------------"
591                 << std::endl;
592
```

```
593       return 1;
594     }
595   catch (...)
596     {
597       std::cerr << std::endl
598                 << std::endl
599                 << "-------------------------------------------------------"
600                 << std::endl;
601       std::cerr << "Unknown exception!" << std::endl
602                 << "Aborting!" << std::endl
603                 << "-------------------------------------------------------"
604                 << std::endl;
605       return 1;
606     }
607
608   return 0;
609 }
610
611 }
```

Code For Deformation using MPI

```
#include <deal.II/base/timer.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/multithread_info.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/sparsity_tools.h>
#include <deal.II/grid/grid_in.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/mpi.h>
#include <deal.II/lac/petsc_vector.h>
#include <deal.II/lac/petsc_sparse_matrix.h>
#include <deal.II/lac/petsc_solver.h>
#include <deal.II/lac/petsc_precondition.h>
#include <deal.II/grid/grid_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <fstream>
#include <iostream>

namespace ProgramMPI
{
  using namespace dealii;

  template <int dim>
  class ElasticProblem
  {
  public:
    ElasticProblem();
    void run();
```

```
46
47  private:
48    void setup_system();
49    void assemble_system();
50    unsigned int solve();
51    void calculate_stress();
52    void refine_grid();
53    void output_results(const unsigned int cycle) const;
54
55    MPI_Comm mpi_communicator;
56
57    const unsigned int n_mpi_processes;
58    const unsigned int this_mpi_process;
59
60    ConditionalOStream pcout;
61
62    GridIn<dim> gridin;
63    Triangulation<dim> triangulation;
64    FESystem<dim> fe;
65    DoFHandler<dim> dof_handler;
66
67    AffineConstraints<double> hanging_node_constraints;
68
69    PETScWrappers::MPI::SparseMatrix system_matrix;
70
71    PETScWrappers::MPI::Vector solution;
72    PETScWrappers::MPI::Vector system_rhs;
73
74    const QGauss<dim> quadrature_formula;
75  };
76
77  template <int dim>
78  const SymmetricTensor<4, dim> ElasticProblem<dim>::stress_strain_tensor =
79    get_stress_strain_tensor<dim>(/*lambda = */ 10.54448e10,
80                                  /*mu = */ 7.6356e10);
81
82  template <int dim>
83  class BodyForceValues : public Function<dim>
84  {
85  public:
86    BodyForceValues();
87
88    virtual void vector_value(const Point<dim> &p,
89                              Vector<double> & values) const override;
90
91    virtual void
92    vector_value_list(const std::vector<Point<dim>> &points,
```

```
                    std::vector<Vector<double>> & value_list) const override;
};


template <int dim>
BodyForceValues<dim>::BodyForceValues()
  : Function<dim>(dim)
{}


template <int dim>
inline void BodyForceValues<dim>::vector_value(const Point<dim> & /*p*/,
                                    Vector<double> &values) const
{
  Assert(values.size() == dim, ExcDimensionMismatch(values.size(), dim));

  const double g = 9.81;
  const double rho = 7850;

  values = 0;
  values(dim - 2) = -rho * g;
}



template <int dim>
void BodyForceValues<dim>::vector_value_list(
  const std::vector<Point<dim>> &points,
  std::vector<Vector<double>> & value_list) const
{
  const unsigned int n_points = points.size();

  Assert(value_list.size() == n_points,
        ExcDimensionMismatch(value_list.size(), n_points));

  for (unsigned int p = 0; p < n_points; ++p)
    BodyForceValues<dim>::vector_value(points[p], value_list[p]);
}

template<int dim>
class PressureBoundaryValues : public Function<dim>
{
  public:
  PressureBoundaryValues(): Function<dim>(1)
  {}

```

```
139     virtual double value(const Point<dim> & p,const unsigned int component = 0)
            ↪ const override;
140
141   };
142
143   template <int dim>
144   double PressureBoundaryValues<dim>::value(const Point<dim> & /*p*/, const
          ↪ unsigned int /*component*/) const
145   {
146     return -1.5e07;//pressure value here
147   }
148
149   template <int dim>
150   ElasticProblem<dim>::ElasticProblem()
151     : mpi_communicator(MPI_COMM_WORLD)
152     , n_mpi_processes(Utilities::MPI::n_mpi_processes(mpi_communicator))
153     , this_mpi_process(Utilities::MPI::this_mpi_process(mpi_communicator))
154     , pcout(std::cout, (this_mpi_process == 0))
155     , fe(FE_Q<dim>(1), dim)
156     , dof_handler(triangulation)
157     , quadrature_formula(fe.degree + 1)
158   {}
159
160   template <int dim>
161   void ElasticProblem<dim>::setup_system()
162   {
163
164     pcout << "setting up... ";
165     GridTools::partition_triangulation(n_mpi_processes, triangulation);
166
167     dof_handler.distribute_dofs(fe);
168     DoFRenumbering::subdomain_wise(dof_handler);
169
170     hanging_node_constraints.clear();
171     DoFTools::make_hanging_node_constraints(dof_handler,
172                                             hanging_node_constraints);
173     hanging_node_constraints.close();
174
175     DynamicSparsityPattern dsp(dof_handler.n_dofs(), dof_handler.n_dofs());
176     DoFTools::make_sparsity_pattern(dof_handler,
177                                     dsp,
178                                     hanging_node_constraints,
179                                     false);
180
181     const std::vector<IndexSet> locally_owned_dofs_per_proc =
182       DoFTools::locally_owned_dofs_per_subdomain(dof_handler);
183     const IndexSet locally_owned_dofs =
```

```
184       locally_owned_dofs_per_proc[this_mpi_process];
185
186     system_matrix.reinit(locally_owned_dofs,
187                          locally_owned_dofs,
188                          dsp,
189                          mpi_communicator);
190
191     solution.reinit(locally_owned_dofs, mpi_communicator);
192     system_rhs.reinit(locally_owned_dofs, mpi_communicator);
193   }
194
195   template <int dim>
196   void ElasticProblem<dim>::assemble_system()
197   {
198     QGauss<dim> quadrature_formula(fe.degree + 1);
199     FEValues<dim> fe_values(fe,
200                             quadrature_formula,
201                             update_values | update_gradients |
202                               update_quadrature_points | update_JxW_values);
203     QGauss<dim-1> face_quadrature_formula(fe.degree +1);
204     FEFaceValues<dim> fe_face_values(fe,
205                             face_quadrature_formula,
206                             update_values | update_normal_vectors|
207                             update_quadrature_points | update_JxW_values);
208
209
210     const unsigned int dofs_per_cell = fe.n_dofs_per_cell();
211     const unsigned int n_q_points = quadrature_formula.size();
212     const unsigned int n_face_q_points = face_quadrature_formula.size();
213
214     FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
215     Vector<double> cell_rhs(dofs_per_cell);
216
217     std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);
218
219     std::vector<double> lambda_values(n_q_points);
220     std::vector<double> mu_values(n_q_points);
221     std::vector<double> pressure_values(n_face_q_points);
222     std::vector<Vector<double>> rhs_values(n_q_points, Vector<double>(dim));
223
224     Functions::ConstantFunction<dim> lambda(7.923e10), mu(11.885e10);
225
226     BodyForceValues<dim> gravity;
227     PressureBoundaryValues<dim> pressure_boundary;
228
229     // store the boundary ids of the surfaces where pressure to be applied
```

```
230     std::vector<int> pressure_boundary_ids =
            ↪ {1,3,4,20,21,22,23,24,25,26,27,28,46,47};
231     // value to be added after physical tag assignation
232     const int add_value = 100;
233     std::transform(pressure_boundary_ids.begin(), pressure_boundary_ids.end(),
234         pressure_boundary_ids.begin(), [add_value](int i) { return i + add_value;
                ↪ });
235
236     for (const auto &cell : dof_handler.active_cell_iterators())
237       if (cell->subdomain_id() == this_mpi_process)
238         {
239           cell_matrix = 0;
240           cell_rhs = 0;
241
242           fe_values.reinit(cell);
243
244           lambda.value_list(fe_values.get_quadrature_points(), lambda_values);
245           mu.value_list(fe_values.get_quadrature_points(), mu_values);
246
247     for (unsigned int i = 0; i < dofs_per_cell; ++i)
248           {
249             const unsigned int component_i =
250               fe.system_to_component_index(i).first;
251
252      for (unsigned int j = 0; j < dofs_per_cell; ++j)
253         {
254           const unsigned int component_j =
255              fe.system_to_component_index(j).first;
256
257         for (unsigned int q_point = 0; q_point < n_q_points;
258                     ++q_point)
259           {
260              cell_matrix(i, j) +=
261                     ((fe_values.shape_grad(i, q_point)[component_i] *
262                       fe_values.shape_grad(j, q_point)[component_j] *
263                       lambda_values[q_point]) +
264                      (fe_values.shape_grad(i, q_point)[component_j] *
265                       fe_values.shape_grad(j, q_point)[component_i] *
266                       mu_values[q_point]) +
267                      ((component_i == component_j) ?
268                         (fe_values.shape_grad(i, q_point) *
269                          fe_values.shape_grad(j, q_point) *
270                          mu_values[q_point]) :
271                         0)) *
272                    fe_values.JxW(q_point);
273                }
274             }
```

```
275           }
276         gravity.vector_value_list(fe_values.get_quadrature_points(), rhs_values)
                ↪ ;
277
278         for (unsigned int i = 0; i < dofs_per_cell; ++i)
279           {
280             const unsigned int component_i =
281               fe.system_to_component_index(i).first;
282
283             for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
284               cell_rhs(i) += fe_values.shape_value(i, q_point) *
285                              rhs_values[q_point](component_i) *
286                              fe_values.JxW(q_point);
287           }
288
289
290
291
292     for(unsigned int face_number =0; face_number < GeometryInfo<dim> ::
            ↪ faces_per_cell;
293                                                          ++face_number)
294         {
295           if (cell->face(face_number)->at_boundary() &&
296           std::find(pressure_boundary_ids.begin(), pressure_boundary_ids.end(),
297                cell->face(face_number)->boundary_id()) != pressure_boundary_ids.
                    ↪ end())
298           {
299             fe_face_values.reinit(cell,face_number);
300             pressure_boundary.value_list(fe_face_values.get_quadrature_points(),
301                                          pressure_values);
302       for(unsigned int i=0; i<dofs_per_cell; ++i)
303           {
304              const unsigned int component_i =
305                fe.system_to_component_index(i).first;
306           for(const unsigned int q_index : fe_face_values.
                  ↪ quadrature_point_indices())
307              {
308                cell_rhs(i) += (pressure_values[q_index] *
309                                fe_face_values.normal_vector(q_index)[component_i])
                                    ↪ *
310                                fe_face_values.shape_value(i,q_index) *
311                                fe_face_values.JxW(q_index);
312              }
313             }
314
315           }
316         }
```

```
317             cell->get_dof_indices(local_dof_indices);
318           hanging_node_constraints.distribute_local_to_global(cell_matrix,
319                                                               cell_rhs,
320                                                               local_dof_indices,
321                                                               system_matrix,
322                                                               system_rhs);
323         }
324
325     system_matrix.compress(VectorOperation::add);
326     system_rhs.compress(VectorOperation::add);
327     FEValuesExtractors::Scalar z_component(dim - 1);
328     std::map<types::global_dof_index, double> boundary_values;
329
330     // store boundary ids of surfaces which are fixed
331     std::vector<int> fixed_boundary_ids = {18,31};
332     // to add this value
333     std::transform(fixed_boundary_ids.begin(), fixed_boundary_ids.end(),
334             fixed_boundary_ids.begin(), [add_value](int i) { return i + add_value;
                ↪ });
335
336     const auto zero_function = Functions::ZeroFunction<dim>(dim);
337
338     for (const auto& boundary_id : fixed_boundary_ids) {
339         VectorTools::interpolate_boundary_values(dof_handler,
340                                                  boundary_id, zero_function,
341                                                  boundary_values);
342     }
343     MatrixTools::apply_boundary_values(
344       boundary_values, system_matrix, solution, system_rhs, false);
345   }
346
347   template <int dim>
348   unsigned int ElasticProblem<dim>::solve()
349   {
350     SolverControl solver_control(2e3, 1e-2);
351     PETScWrappers::SolverCG cg(solver_control, mpi_communicator);
352
353     PETScWrappers::PreconditionBlockJacobi preconditioner(system_matrix);
354
355     cg.solve(system_matrix, solution, system_rhs, preconditioner);
356           Vector<double> localized_solution(solution);
357
358     hanging_node_constraints.distribute(localized_solution);
359     return solver_control.last_step();
360   }
361
362   template <int dim>
```

```
363   void ElasticProblem<dim>::refine_grid()
364   {
365     const Vector<double> localized_solution(solution);
366
367     Vector<float> local_error_per_cell(triangulation.n_active_cells());
368     KellyErrorEstimator<dim>::estimate(dof_handler,
369                                        QGauss<dim - 1>(fe.degree + 1),
370                                        {},
371                                        localized_solution,
372                                        local_error_per_cell,
373                                        ComponentMask(),
374                                        nullptr,
375                                        MultithreadInfo::n_threads(),
376                                        this_mpi_process);
377
378     const unsigned int n_local_cells =
379       GridTools::count_cells_with_subdomain_association(triangulation,
380                                                         this_mpi_process);
381     PETScWrappers::MPI::Vector distributed_all_errors(
382       mpi_communicator, triangulation.n_active_cells(), n_local_cells);
383
384     for (unsigned int i = 0; i < local_error_per_cell.size(); ++i)
385       if (local_error_per_cell(i) != 0)
386         distributed_all_errors(i) = local_error_per_cell(i);
387     distributed_all_errors.compress(VectorOperation::insert);
388
389
390     const Vector<float> localized_all_errors(distributed_all_errors);
391
392     GridRefinement::refine_and_coarsen_fixed_number(triangulation,
393                                                     localized_all_errors,
394                                                     0.3,
395                                                     0.03);
396     triangulation.execute_coarsening_and_refinement();
397   }
398
399
400 template <int dim>
401 void ElasticProblem<dim>::output_results(const unsigned int cycle) const
402   {
403     const Vector<double> localized_solution(solution);
404
405     if (this_mpi_process == 0)
406       {
407         std::ofstream output("37MnSi7 yo ho -" + std::to_string(cycle) + ".vtk");
408         std::vector<DataComponentInterpretation::DataComponentInterpretation>
409           data_component_interpretation(
```

```
410               dim, DataComponentInterpretation::component_is_part_of_vector);
411       DataOut<dim> data_out;
412       data_out.attach_dof_handler(dof_handler);
413
414       std::vector<std::string> solution_names(dim, "Displacement");
415       data_out.add_data_vector(localized_solution,
416                                solution_names,
417                                DataOut<dim>::type_dof_data,
418                                data_component_interpretation);
419
420       std::vector<unsigned int> partition_int(triangulation.n_active_cells());
421       GridTools::get_subdomain_association(triangulation, partition_int);
422
423       const Vector<double> partitioning(partition_int.begin(),
424                                         partition_int.end());
425
426       data_out.add_data_vector(partitioning, "partitioning");
427       data_out.build_patches();
428       data_out.write_vtk(output);
429     }
430   }
431
432 template <int dim>
433   void ElasticProblem<dim>::run()
434   {
435     Timer timer;
436     for (unsigned int cycle = 0; cycle < 1; ++cycle)
437       {
438         pcout << "Cycle "<< cycle << ':' << std::endl;
439
440         if (cycle == 0)
441           {
442             // import the mesh file generated by Gmesh
443             gridin.attach_triangulation(triangulation);
444             std::ifstream f("Mesh.msh");
445             gridin.read_msh(f);
446           }
447         else
448           refine_grid();
449
450         pcout << "Number of active cells: "
451               << triangulation.n_active_cells() << std::endl;
452
453         setup_system();
454
455         pcout << "Number of degrees of freedom:" << dof_handler.n_dofs()
456               << "(by_partition:";
```

```
457         for (unsigned int p = 0; p < n_mpi_processes; ++p)
458           pcout << (p == 0 ? '': '+')
459                 << (DoFTools::count_dofs_with_subdomain_association(dof_handler,
460                                                                     p));
461         pcout << ')' << std::endl;
462 
463         assemble_system();
464         const unsigned int n_iterations = solve();
465 
466         pcout << "Solver converged in "<< n_iterations << "iterations."
467               << std::endl;
468 
469         output_results(cycle);
470         pcout<< "Time:" << timer.cpu_time() << "sec." << std:: endl;
471       }
472   }
473 } // namespace ProgramMPI
474 
475 
476 
477 int main(int argc, char **argv)
478 {
479   try
480     {
481       using namespace dealii;
482       using namespace ProgramMPI;
483 
484       Utilities::MPI::MPI_InitFinalize mpi_initialization(argc, argv, 1);
485 
486       ElasticProblem<3> elastic_problem;
487       elastic_problem.run();
488     }
489   catch (std::exception &exc)
490     {
491       std::cerr << std::endl
492                 << std::endl
493                 << "----------------------------------------------------"
494                 << std::endl;
495       std::cerr << "Exception on processing: "<< std::endl
496                 << exc.what() << std::endl
497                 << "Aborting!" << std::endl
498                 << "----------------------------------------------------"
499                 << std::endl;
500 
501       return 1;
502     }
503   catch (...)
```

```
504     {
505       std::cerr << std::endl
506                 << std::endl
507                 << "------------------------------------------------------"
508                 << std::endl;
509       std::cerr << "Unknown exception!" << std::endl
510                 << "Aborting!" << std::endl
511                 << "------------------------------------------------------"
512                 << std::endl;
513       return 1;
514     }
515
516   return 0;
517 }
```

APPENDIX C: SCRIPT FOR GEOMETRY SIMPLIFICATION AND MESHING

```
// This script creates simplified cylinder model and mesh it.
//lc means target mesh size
lc = 1e-2;

//defining points

Point(20) = {0, 0, 0, lc};
Point(2) = {0, 5.6, 0, lc};
Point(3) = {86, 0, 0, lc};
Point(4) = {86, 5.6, 0, lc};
Point(5) = {86, 30, 0, lc};
Point(6) = {110.4, 30, 0, lc};
Point(7) = {116, 30, 0, lc};
Point(8) = {110.4, 1220, 0, lc};
Point(9) = {116, 1220, 0, lc};
Point(10) = { 0, 1220, 0, lc};
Point(11) = {15, 1329.38, 0, lc};
Point(12) = {15, 1335.03, 0, lc};
Point(13) = {65.05, 1309.2, 0, lc};

//lines

Line(1) = {2, 20};
Line(2) = {3, 20};
Line(3) = {2, 4};
Line(4) = {6, 8};
Line(5) = {7, 9};
Line(6) = {11, 12};

//Circles

//+
Circle(8) = {11, 10, 8};
//+
Circle(9) = {12, 10, 9};
//+
Circle(10) = {4, 5, 6};
//+
Circle(11) = {3, 5, 7};

//curve loop

Physical Curve("1", 12) = {1, 3, 10, 4, 8, 6, 9, 5, 11, 2};
Curve Loop(1) = {4, -8, 6, 9, -5, -11, 2, -1, 3, 10};
```

```
45 Plane Surface(1) = {1};
46
47 //Transfinite Surface {1};
48 //Recombine Surface {1};
49
50 Extrude {{0, 1, 0}, {0, 5.6,0}, 0.5*Pi} {
51 Surface{1}; Layers{6};
52 Recombine;
53 }
54
55 //+
56 Extrude {{0, 1, 0}, {0, 0, 0}, Pi/2} {
57   Surface{59}; Layers{8}; Recombine;
58 }
59 //+
60 Extrude {{0, 1, 0}, {0, 0, 0}, Pi/2} {
61   Surface{106}; Layers{8}; Recombine;
62 }
63 //+
64 Extrude {{0, 1, 0}, {0, 0, 0}, Pi/2} {
65   Surface{153}; Layers{8}; Recombine;
66 }
67
68
69 Mesh.Algorithm = 6;
70 Mesh.ElementOrder = 1;
71 Mesh 3;
72 Coherence Mesh;
```

```
  //This code take step file as input and takes
  its cross section first and revolve that
  cros section with extruding mesh
   SetFactory("OpenCASCADE");
v() = ShapeFromFile("file.STEP");
// Get the bounding box of the volume:
bbox() = BoundingBox Volume{v()};
xmin = bbox(0);
ymin = bbox(1);
zmin = bbox(2);
xmax = bbox(3);
ymax = bbox(4);
zmax = bbox(5);

//Defining Necessary Parameter
dx = (xmax - xmin);
dy = (ymax - ymin);
dz = (zmax - zmin);
L =dz/2 ;
H = dy;

//Define Cutting Surface
s() = {news};
Rectangle(s(0)) = {xmin, ymin, zmin, L, H};
Rotate{ {0, 1, 0}, {xmin, ymin, zmin}, -Pi/2 }
{ Surface{s(0)}; }
tx = dx / 2;
ty =0;
tz=0;
Translate{tx, ty, tz} { Surface{s(0)}; }


//Delete Everything of surface out of object
i.e keeping cutting surface inside material domain only
BooleanFragments{ Volume{v()}; Delete; }
{ Surface{s()}; Delete; }
Recursive Delete { Surface{:}; }

//Deleting everything except the surface of cut
  eps = 1e-4;
  s() = {};
    xx = xmin;
    yy = ymax;
    zz = zmax;
    s() += Surface In BoundingBox
    {xmin - eps + tx, ymin - eps +ty, zmin - eps + tz,
```

```
 xx + eps + tx, yy + eps + ty, zz + eps +tz};
  dels = Surface{:};
  dels -= s();
  Delete { Volume{:}; Surface{dels()};
  Curve{:}; Point{:}; }

//+
Extrude {{0, 1, 0}, {0, 0, 0}, Pi/2} {
  Surface{2}; Layers{8}; Recombine;
}
//+
Extrude {{0, 1, 0}, {0, 0, 0}, Pi/2} {
  Surface{12}; Layers{8}; Recombine;
}
//+
Extrude {{0, 1, 0}, {0, 0, 0}, Pi/2} {
  Surface{22}; Layers{8}; Recombine;
}
//+

Extrude {{0, 1, 0}, {0, 0, 0}, Pi/2} {
  Surface{32}; Layers{8}; Recombine;
}
Coherence;
Coherence Mesh;
//+
//+
```

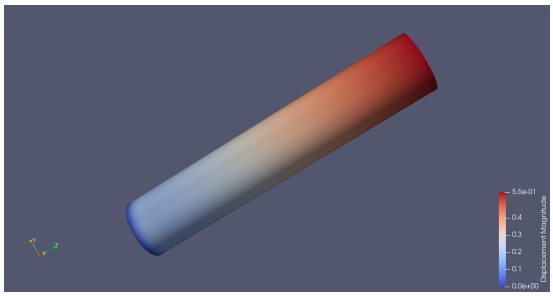
APPENDIX D: SCRIPT FOR BOUNDARY ID ASSIGNING

```
with open('inputmesh.txt','r') as infile,
open('outputmesh.txt','w') as outfile:
switch = 0
turn = 0
for line in infile:
    # Code to modify line
    columns = line.split(' ')
    # store the data in columns as strings
    if columns[0] == '$Elements\n':
    # when elements are found
        switch = 1 # turn on this switch
        print(columns)
    if switch == 1:
 # start reading the lines after Elements are found
     if len(columns)>1 and columns[1] != '15':
 # when the line starts containing other than 15
         turn = 1
      switch = 0
 # no need of this section after assigning turn is 1
    if turn == 1 and columns[0] != '$EndElements\n':
 # before reaching the endelement section
        tmp = int(columns[4]) + 100
        # assign the physical ids
        columns[3] = str(tmp)
        new_line = (' ').join(columns)
        # create new line after the manipulation
    else:
        new_line = line
    outfile.write(new_line)
    #write the line in new file
    # print(columns)
```

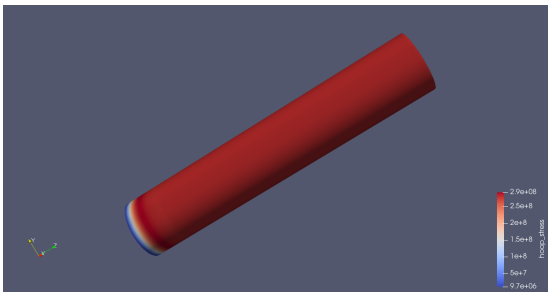
APPENDIX E: ADDITIONAL FEM RESULTS

Simulation results for hollow cylinder from deal.ii

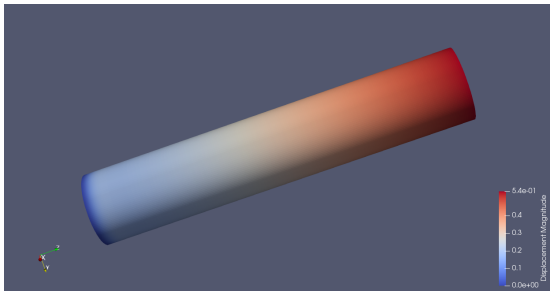
Material : 37MnSi5



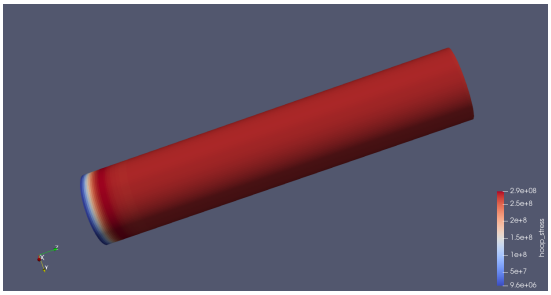
(a) Displacement, Thickness=5.4



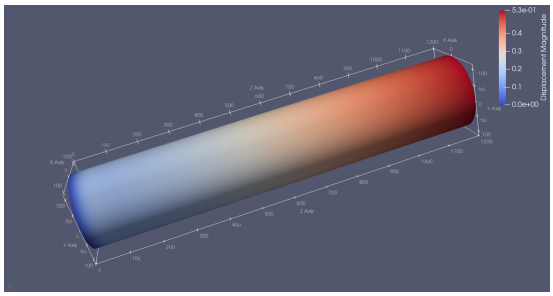
(b) Hoop stress, Thickness=5.4



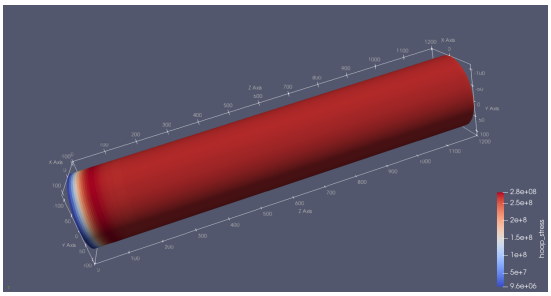
(c) Displacement, Thickness=5.5



(d) Hoop stress, Thickness=5.5



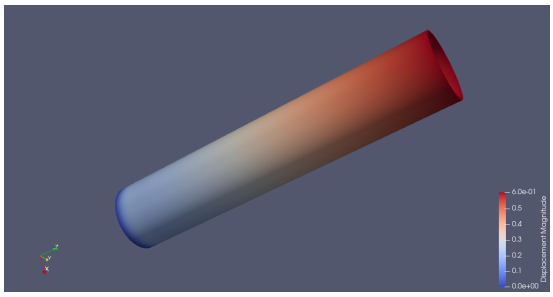
(e) Displacement, Thickness=5.6



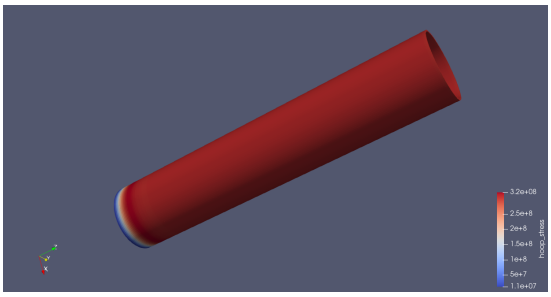
(f) Hoop stress, Thickness=5.6

Figure E.1: Displacement and Hoop stress with different thickness (37MnSi5)

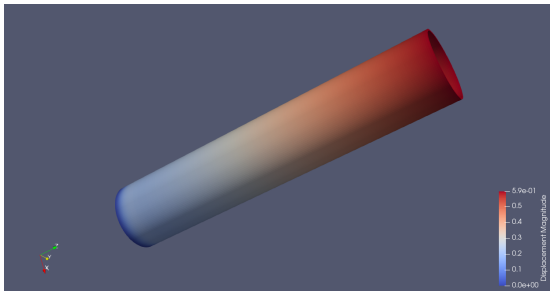
Material : 34Mn2V



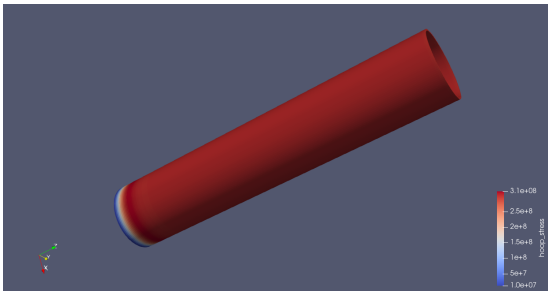
(a) Displacement, Thickness=5.4



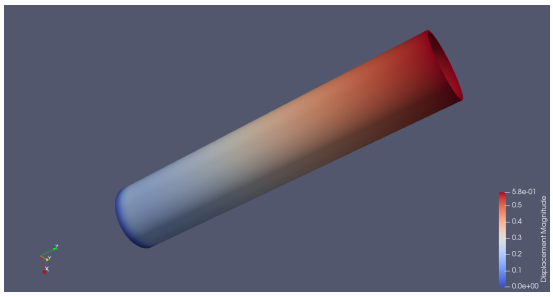
(b) Hoop stress, Thickness=5.4



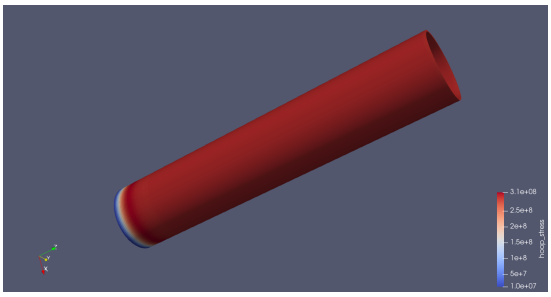
(c) Displacement, Thickness=5.5



(d) Hoop stress, Thickness=5.5



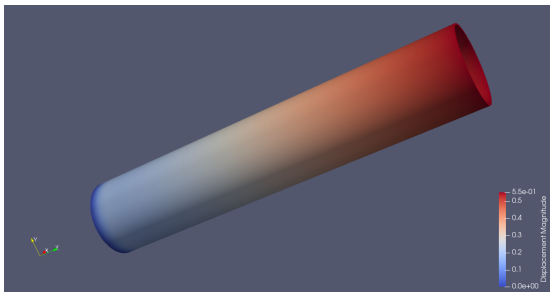
(e) Displacement, Thickness=5.6



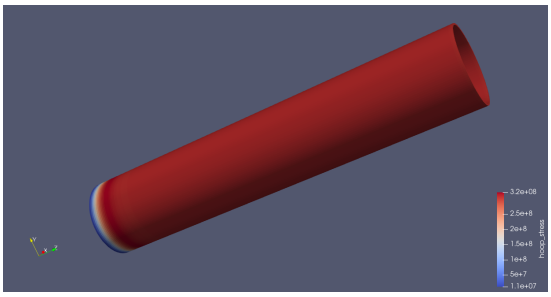
(f) Hoop stress, Thickness=5.6

Figure E.2: Displacement and Hoop stress with different thickness (34Mn2V)

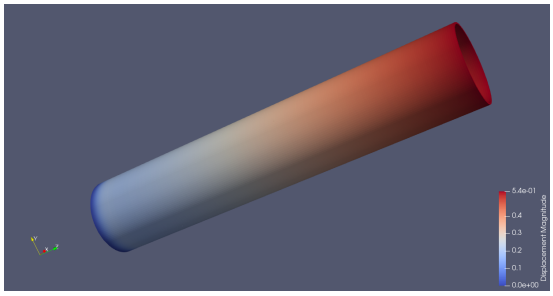
Material : 32CrMo4



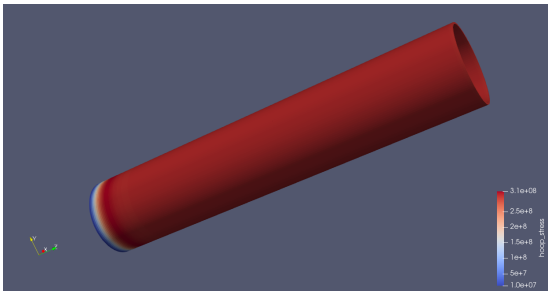
(a) Displacement, Thickness=5.4



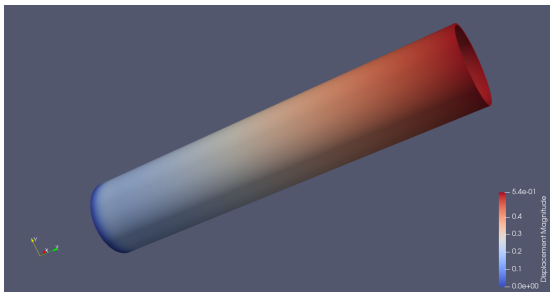
(b) Hoop stress, Thickness=5.4



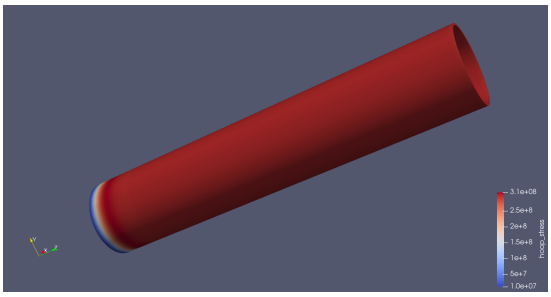
(c) Displacement, Thickness=5.5



(d) Hoop stress, Thickness=5.5



(e) Displacement, Thickness=5.6



(f) Hoop stress, Thickness=5.6

Figure E.3: Displacement and Hoop stress with different thickness (32CrM04)