Master's thesis

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

Aleksander Hatlestad Elmer
Brage Lund Aakre
Isak Flo Bødal

Exploring Finite Element Analysis with Higher-order Solid Elements in Algorithms Aided Design and Optimization with Help of Machine Learning

Master's thesis in Civil and Environmental Engineering
Supervisor: Associate Professor Marcin Luczkowski
Co-supervisor: Dr. techn. Konstantinos Gavriil
June 2022

NTNU
Norwegian University of
Science and Technology

Aleksander Hatlestad Elmer
Brage Lund Aakre
Isak Flo Bødal

Exploring Finite Element Analysis with Higher-order Solid Elements in Algorithms Aided Design and Optimization with Help of Machine Learning

Master's thesis in Civil and Environmental Engineering
Supervisor: Associate Professor Marcin Luczkowski
Co-supervisor: Dr. techn. Konstantinos Gavriil
June 2022

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

NTNU
Norwegian University of
Science and Technology

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

ACCESSIBILITY

OPEN

MASTER THESIS 2022

SUBJECT AREA:	DATE:	NO. OF PAGES:
Structural Engineering	11/06/2022	vii + 116

TITLE:
Exploring Finite Element Analysis with Higher-order Solid Elements in Algorithms Aided Design and Optimization with Help of Machine Learning

Utforsking av Elementanalyse med Høyere-ordens Volumelementer i Algorithms Aided Design og Optimalisering med Maskinlæring

BY:
Aleksander Hatlestad Elmer
Brage Lund Aakre
Isak Flo Bødal

SUMMARY:
This thesis explores the possibilities of Finite Element Analysis (FEA) with solid elements in an Algorithms Aided Design (AAD) environment. An AAD environment is a program that uses an algorithmic approach with parameters to create 3D models. In this thesis, the AAD program used is Grasshopper. Furthermore, the advantages of applying Machine Learning (ML) to some of the more challenging steps when implementing the FEA in Grasshopper has been investigated. ML is applied to make parts of the analysis more efficient, or to solve problems that are too complex to solve in a satisfying manner with a program that uses fixed rules.

Two different finite element plugins for Grasshopper are developed. The first is fully developed by the authors of this thesis, while the second plugin is a further development of an already established plugin. These plugins illustrate how the implementation of FEA in Grasshopper makes the design development process more efficient. One of the greatest benefits of working in Grasshopper is the ability to rapidly make changes to the geometry. The plugins developed in this thesis improve the design process by adding the option to perform an FEA with solid elements. The second plugin also highlights the benefits of implementing higher-order elements in an AAD environment, where we often find more complex geometry.

In addition, a series of ML models are developed to answer the second research question. One of the most challenging steps in the FEA is preprocessing of the model, especially meshing of complex geometry has proven to be challenging. Therefore, the meshing process is one of the tasks investigated with the help of ML. The thesis also explores if ML can be used to replace the entire FEA, and if it can be used to predict the rotational stiffness of a connection.

All the ML models included in this thesis show promising results. The problem of finding the location of mesh vertices is solved for both 2D and 3D. To create complete meshes, the method would need more development. For the entire FEA problem, the results show accurate predictions for stresses, while the predicted displacement needs some further improvement before the results are satisfactory.

In general, it is clear that the use of ML in the FEA workflow will improve the efficiency and accuracy of the analysis. Furthermore, the implementation of the FEA illustrates the possibilities of improving the rapid design process in an AAD environment.

RESPONSIBLE TEACHER: Associate Professor Marcin Luczkowski

SUPERVISOR(S): Associate Professor Marcin Luczkowski and Dr. techn. Konstantinos Gavriil

CARRIED OUT AT: Department of Structural Engineering, Norwegian University of Science and Technology

Preface

This thesis concludes our Master of Science degree in Structural Engineering at the department of Structural Engineering at the Norwegian University of Science and Technology.

We would like to thank our supervisor Marcin Luczkowski and co-supervisors Konstantinos Gavrill and Arturs Berzins for their help and guidance during the course of this theis. We would also like to thank our friends and family for their support during all these years.

Aleksander Hatlestad Elmer
Brage Lund Aakre
Isak Flo Bødal

Abstract

This thesis explores the possibilities of Finite Element Analysis (FEA) with solid elements in an Algorithms Aided Design (AAD) environment. An AAD environment is a program that uses an algorithmic approach with parameters to create 3D models. In this thesis, the AAD program used is Grasshopper. Furthermore, the advantages of applying Machine Learning (ML) to some of the more challenging steps when implementing the FEA in Grasshopper has been investigated. ML is applied to make parts of the analysis more efficient, or to solve problems that are too complex to solve in a satisfying manner with a program that uses fixed rules. The thesis consists of seven case studies designed to explore the following research questions.

What are the possibilities and issues of Finite Element Analysis with solid elements in an Algorithms Aided Design environment?

Can parts of, or the whole, Finite Element Analysis be replaced by machine learning?

To answer the first research question, two different finite element plugins for Grasshopper are developed. The first is fully developed by the authors of this thesis, while the second plugin is a further development of an already established plugin. These plugins illustrate how the implementation of FEA in Grasshopper makes the design development process more efficient. One of the greatest benefits of working in Grasshopper is the ability to rapidly make changes to the geometry. The plugins developed in this thesis improve the design process by adding the option to perform an FEA with solid elements. The second plugin also highlights the benefits of implementing higher-order elements in an AAD environment, where we often find more complex geometry.

A series of ML models are developed to answer the second research question. One of the most challenging steps in the FEA is preprocessing of the model, especially meshing of complex geometry has proven to be challenging. Therefore, the meshing process is one of the tasks investigated with the help of ML. The thesis also explores if ML can be used to replace the entire FEA, and if it can be used to predict the rotational stiffness of a connection.

All the ML models included in this thesis show promising results. The problem of finding the location of mesh vertices is solved for both 2D and 3D. To create complete meshes, the method would need more development. For the entire FEA problem, the results show accurate predictions for stresses, while the predicted displacement needs some further improvement before the results are satisfactory.

In general, it is clear that the use of ML in the FEA workflow will improve the efficiency and accuracy of the analysis. Furthermore, the implementation of the FEA illustrates the possibilities of improving the rapid design process in an AAD environment.

Sammendrag

Denne masteroppgaven utforsker mulighetene ved å implementere Elementsanalyse med volumelementer i program som benytter seg av Algorithms Aided Design (AAD). AAD er en metode for å bruke algoritmer og parametere til å lage 3D modeller. I denne oppgaven blir programmet Grasshopper brukt. Videre ser oppgaven på å erstatte de mer komplekse delene av elementanalysen med Maskinlæring (ML). ML blir brukt for å effektivisere deler av analysen, eller til å løse oppgaver som er for komplekse til å bli løst av en algoritme med faste regler. Oppgaven består av syv case studier som utforsker følgende problemstillinger.

Hva er mulighetene og utfordringene ved å implementere elementanalyse med volumetriske elementer i et Algorithms Aided Design program?

Kan deler av, eller hele, analysen erstattes med maskinlæring?

For å svare på den første problemstillingen har to ulike programvareutvidelser til Grasshopper, som gjennomfører elementanalyse, blitt utviklet. Den første av disse er utviklet i sin helhet av oppgavens forfattere, mens den andre er en videreutvikling av en allerede etablert programvare. Disse to programvarene viser hvordan implementeringen av elementanalyse i AAD effektiviserer designprosessen. En av de største fordelene ved å jobbe i Grasshopper er muligheten til å raskt gjøre endringer på geometrien til en modell. Den andre programvareutvidelsen som ble utviklet viser fordelene ved å bruke elementer med høyere-ordens formfunksjoner i analysen. Dette spesielt på kompleks geometri, som er vanlig å lage i AAD programmer.

Videre ble det utviklet en rekke ML modeller for å svare på den andre problemstillingen. En av de største utfordringene i elementanalysen er å lage et godt mesh, spesielt for kompleks geometri. Det har derfor blitt utforsket å bruke ML til å forbedre denne prosessen. I tillegg har det blitt sett på å erstatte hele elementanalysen med ML, samt å bruke ML til å predikere rotasjonsstivheten til en forbindelse.

Alle ML-modellene i denne oppgaven gir positive resultater. Oppgaven med å finne plasseringene til nodene i meshet er løst både i 2D og i 3D. For å kunne generere ett fullstendig mesh, må metoden utvikles ytterligere. Resultatene når ML blir brukt på hele elementanalysen er gode for spenninger, mens det for deformasjoner kreves forbedring før resultatene er tilfredsstillende.

Det er tydelig at å bruke ML til å gjennomføre deler av, eller hele, elementanalysen vil forbedre effektiviteten. I tillegg viser implementeringen av elementanalyse som støtter volumetriske elementer hvordan designprosessen i AAD program kan forbedres og effektiviserer.

Glossary

AAD	Algorithms Aided Design is a method for creating digital models with an algorithm based-approach.
Artificial Intelligence	The theory and development of computer systems able to perform tasks normally requiring human intelligence.
Batch size	Number of training samples utilized in one iteration.
Convolutional network	A network architecture specifically designed to process pixel data.
Deep learning	A neural network with two or more hidden layers.
DF	Distance Field, a field that describe the distance to an unclosed object.
Dropout	Data that is intentionally dropped from the neural network to improve processing and time to results.
Fully connected network	A network architecture where all neurons in two subsequent layers are connected.
Epochs	The number of times a learning algorithm runs through the whole dataset.
FEA	Finite Element Analysis is an analysis using the Finite Element Method, which is a method numerical calculation method.
Feature	The input property for the algorithm.
Hidden layer	A layer between input and output layers that contains neurons.
Learning rate	A tuning parameter that determines the step size of each iteration.
Loss function	A function that calculates the error between the prediction and the target.
Machine Learning	A branch of AI that uses large datasets to imitate the way humans learn.
Neural Networks	A series of algorithms that aims to recognize the relationship between a set of features and a set of targets.
Neurons	A connection point in a neural network that imitates the neurons in a human brain.
SDF	Sign Distance Field, a field that describe the distance to a closed object.
Target or label	The target for an algorithm to predict.
Unet	A network architecture designed for image segmentation and classification.
Weight	A weight controls the strength of the connection between two neurons. I.e., it decides how much a neuron influences the output.

Table of Contents

1 Introduction

This thesis was written for the Conceptual Structural Design Group (CSDG) of NTNU. The CSDG explores the relationship between structural engineering and architectural design. Conceptual structural design is about creating designs that in addition to carrying loads, also appear meaningful, beautiful and otherwise interesting. In this thesis, the aim is to improve the structural design process and enhance the possibilities of creating interesting structures.

1.1 Background

In the traditional workflow of a building project the architect is responsible for the design, while the engineer is responsible for calculating dimensions and controlling the architects design solutions. If something does not work, the engineer sends the design back to the architect. The design goes back and forth until both the architect and the engineer are satisfied. The design is then sent to the cost calculation and finally to the construction contractor. This workflow is ineffective and gives an increase in cost. Figure 1.1 shows the cost of making design changes, together with the ability to implement them, in terms of time. As the project progresses, the cost of making changes increases, while the ability to implement these changes decreases. If the design gets stuck in a loop between the architect and the engineer, the cost of making a small change increases drastically.

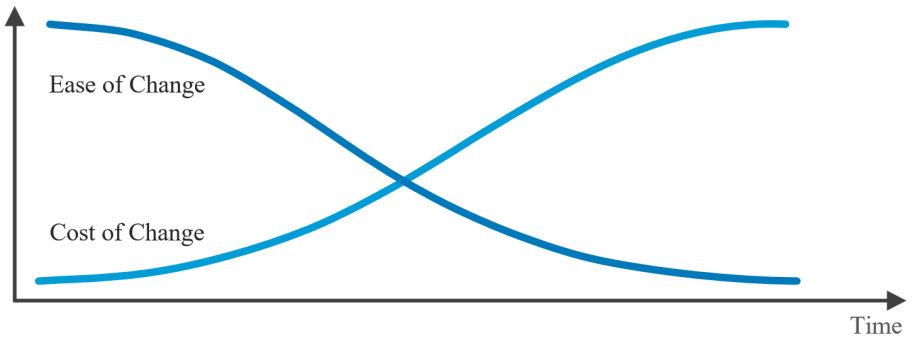


Figure 1.1: Relationship between cost and ability of making a change, and the time development of a project (Eikeland, 2001).

A solution to the flaws in what has been described as the traditional workflow of a building project is *Digital Prototyping*. Digital prototyping is a technique to quickly create a digital computer-aided design (CAD) model. The model can be easily changed to create multiple design proposals in a short amount of time. By creating an easily adjustable CAD model early in the design process, the engineer and architect could work together to create design proposals. In this way, the design is driven not only by aesthetics, but also by good engineering solutions, which leads to better designs. The main bottleneck in the process is that the prototype models must be exported or recreated in a dedicated FEA program. These programs are computationally demanding to run and the analyzes are often slow. This, in turn, increases the cost and decreases the effectiveness of the design process.

A method for creating digital prototypes, called *Algorithms Aided Design* (AAD), is becoming increasingly popular. When creating a model in AAD, the traditional graphical representation is replaced by an algorithm, often done through visual programming. The algorithmic approach to creating geometry allows for the parameters that determine the model to be changed in real time. This makes it quick to produce a lot of new design proposals. When moving from manual design exploration to an algorithm-based approach, it would be natural to explore the possibility of linking the FEA part of the design process with the AAD part. The application of the Finite Element Method (FEM) in an AAD environment would make the whole design process algorithm-based. This, in turn, would streamline and make the digital prototyping process more efficient.

For AAD based FEA, there is already some established software. These are mainly based on beam and shell elements, which works well for the main parts of a structure: the beams, columns, walls, roofs, etc. The connections between these parts are more problematic, because it is not possible to represent a connection correctly with beam- or shell elements. To analyze these, an analysis with solid elements is required. FEA with solid elements are computationally costly, especially when working with higher-order shape functions. When working with AAD, it is desirable to have everything updated in real time. It is not effective if parts of the model must be remodeled in another FEA program for every small change to the design. Additionally, it is often difficult to create a mesh of high enough quality, especially for more complex geometry. The idea to use *Artificial Intelligence* (AI) to accelerate the FEA process and improve the mesh generation is therefore explored.

Machine Learning (ML) is a subcategory of AI and creates algorithms based on large sets of data. It is often used at tasks that are too complex or difficult to solve for human written programs. The algorithm uses the provided data to learn, instead of using an algorithm with fixed rules. This could be a solution to the computational cost problems described. Since ML uses large datasets to create an algorithm to solve problems, it should be able to replace an FEM solver in the AAD environment. This would make it possible to create design proposals with a good approximation for the FEA results in real time.

1.2 Research Question

This thesis aims to answer the following research questions:

What are the possibilities and issues of finite element analysis with solid elements in an AAD environment?

Can parts of, or the whole, finite element analysis be replaced by machine learning?

By exploring these questions, we hope to find a way to improve the workflow and efficiency of the conceptual design phase. By implementing FEA and ML software to improve the efficiency of the FEA in AAD software such as Grasshopper, the design process will be more efficient. This, in turn, will improve the cooperation between architects and engineers in the design phase, with both sides being able to influence design choices more easily. Furthermore, it would create the possibilities to analyze more complex structural systems, like a grid of connections.

To answer these questions, an FEM plug-in with solid elements has to be developed for Grasshopper. This solver needs to support elements with higher-order shape functions to be applicable for detailed design of connections, because these elements better represent complex geometry. Furthermore, an ML model needs to be developed to enhance different parts of the FEA. It would be sensible to explore the application of ML to both the meshing and the full analysis. Meshing of complex and nonuniform geometry is difficult and would greatly improve an analysis if this were done by ML.

1.3 Structure of the Thesis

This thesis is based on case studies that explore the research questions. It begins with an introduction to the relevant theory used for software development. This includes an introduction to FEM and ML, with different network architectures and algorithm building. There is also a sub-chapter on rotational stiffness. A chapter describing relevant third-party software follows. Chapter 4 gives an in-depth explanation of the structure and architecture of the different software developed by the authors of this thesis. This includes the simple *FEM Solver* plug-in for Grasshopper, the further development of the *SolidFEM* plug-in for Grasshopper, as well as the framework for creating ML models in *PyTorch*. All source codes can be found in the appendix, in Table A.1. The different case studies are presented in Chapter 5, each case study being described in detail, along with a result and a small discussion section. Lastly, there is a discussion and conclusion section that connects all parts of the thesis together with the research questions.

2 Theory

2.1 Finite Element Method

2.1.1 Introduction

FEM is a well-established numerical method that aims to find approximate solutions to load responses. Since the method works by dividing complex systems into smaller and more manageable subsystems, it is often used for structures with advanced geometries that are impossible to describe with analytical formulations. These subsystems are commonly known as elements, and these elements are a representation of the geometry of the model. There are four main elements: beam, shell, plate, and solids. This thesis focuses on solid elements. The following theory will explain the general concepts of FEM, before the theory for solid elements is explained with the help of an eight-noded hexahedron element.

Assumptions

To describe the physical world in terms of mathematical formulations, some simplification is needed. The theory is based on elastic and homogeneous materials. Furthermore, the load response is calculated using linear theory. Linear theory is based on two main assumptions (Bell, 2013):

- The calculated displacements are assumed to be small. Therefore, the equilibrium calculations can be based on an undeformed geometry.
- The material is assumed to be linear elastic. Such that the relationship between stress and strain is linear and reversible.

This thesis formulates the relationship between force, displacement, and stress using the *Principle of Virtual Work*, (Bell, 2013). The principle assumes a small *virtual displacement*, which introduces changes to the system. Furthermore, it assumes that the work performed by a real external force is in equilibrium with the work performed by internal forces, both as a result of the virtual displacements, for a system in static equilibrium.

2.1.2 FEM procedure

Discretization

As mentioned in the introduction to the theory, the geometry of a model must be dicretizised into smaller elements to perform an FEA on it. These elements consist of a set of *nodes*, and the number of nodes per element depends on the type of element. Elements are connected through neighboring nodes, creating the models *connectivity*. This ensures that the elements are *compatible*, that is, that the entire model is connected without any overlap or gaps.

Load - Displacement relation

The basis of FEM theory is the relationship between loads and displacements. Nodal displacements are calculated in Eq. 2.1.

$$\mathbf{Kr} = \mathbf{R} \tag{2.1}$$

In this equation, **K** represents *global stiffness matrix*, **r** is a vector of the unknown *nodal displacements*, and **R** is a vector of the applied loads at the nodes. By calculating *nodal displacements*, stresses and strains can be calculated.

Shape Functions

To describe the displacements within the elements, the nodal displacements are interpolated. The interpolation is performed with *shape functions*. The construction of these shape functions is based on a set of principles that must be followed. These are *continuity* and *completeness*, as defined by (Bell, 2013).

- The displacements are required to be continuous over all elements. If the highest-order differential equation in FEM is denoted by m, then the shape functions and its derivatives should be continuous up to and including m-1.
- The completeness principle requires the shape functions to correctly describe the *rigid body* movements of the elements. This is achieved when a rigid body movement does not cause any stress in the element. Furthermore, they must be able to represent a state of constant stress within the element.

The shape functions are denoted as **N** and determine the variation of displacements within an element. It is given by Eq. 2.2 and Eq. 2.3.

$$\mathbf{N}(x,y,z) = \begin{bmatrix} \mathbf{N}_1(x,y,z) & \mathbf{N}_2(x,y,z) & \dots & \mathbf{N}_{n_d}(x,y,z) \end{bmatrix} \tag{2.2}$$

where

$$\mathbf{N}_i = \begin{bmatrix} N_{i1} & 0 & \cdots & 0 \\ 0 & N_{i2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & N_{in_f} \end{bmatrix} \tag{2.3}$$

Here, n_d is the total number of nodes, while n_f is the total number of *degrees of freedom* (DOFs). Which for a model based on solid elements will be three *DOFs* per node, since the solid elements only account for translation and not rotation.

The displacements are calculated by

$$\mathbf{u}(x,y,z) = \sum_{i=1}^{n_d} \mathbf{N}_i(x,y,z)\mathbf{r}_i = \mathbf{N}(x,y,z)\mathbf{r}_e \tag{2.4}$$

with $\mathbf{r}_i$ being the displacements of node i for the three DOFs.

$$\mathbf{r}_i = \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} \tag{2.5}$$

The total number of DOFs for an element is equal to $n_f \cdot n_d$. As mentioned above, $n_f = 3$ for solid 3D elements. Furthermore, the shape functions are the same for each DOF: $N_i = N_{i1} = N_{i2} = N_{i3}$.

Stiffness Matrix

The local stiffness matrix of an element can be defined from the principle of virtual work as

$$\mathbf{k}_e = \int_{V_e} \mathbf{B}^T \mathbf{c}\mathbf{B}\, dV \tag{2.6}$$

where $\mathbf{c}$ is the material matrix and $\mathbf{B}$ is the strain matrix. $\mathbf{B}$ is defined as the derivative of the shape functions $\mathbf{B} = \Delta\mathbf{N}$.

Since the local stiffness matrix is related only to one element, it is necessary to assemble all local stiffness matrices into a global stiffness matrix. This is done by constructing the connectivity matrix $\mathbf{a}$ for each element. The $\mathbf{a}$ matrix relates the local position of the nodes to their global position; see Eq. 2.7.

$$\mathbf{K} = \sum_{k=1}^{n_e} \mathbf{a}_e^T \mathbf{k}_e \mathbf{a}_e \tag{2.7}$$

Boundary conditions

Boundary conditions are constraints on nodal displacements. A fixed node means that the nodal DOFs are predefined to be 0. To simplify the calculations and make the analysis more efficient, this can be included in the load-displacement relationship given in Eq. 2.1. The global stiffness matrix and the load vector can be reduced in correspondence with the fixed DOFs, as illustrated in Eq. 2.8.

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{21} & K_{22} & K_{23} & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & K_{43} & K_{44} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ 0 \\ r_4 \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \rightarrow \begin{bmatrix} K_{11} & K_{12} & K_{14} \\ K_{21} & K_{22} & K_{24} \\ K_{41} & K_{42} & K_{44} \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ r_4 \end{bmatrix} = \begin{bmatrix} R_1 \\ R_2 \\ R_4 \end{bmatrix} \tag{2.8}$$

Loads

External loads can be applied by either point loads directly in the nodes or by surface loads. The external point loads are deconstructed and assigned to the corresponding DOFs to construct the load vector **R**. For distributed surface loads, there are different approaches to *lumping* these at the nodes. Lumping is a method of distributing surface loads to nodal loads. The simplest method is to multiply the distributed load by the area of the surface and divide it by the number of nodes. When going element by element, the nodes connected to several elements get a contribution from all. The other method creates a consistent nodal load vector. This method is based on the idea that the external work, **W**, done by nodal loads $\mathbf{S}^0$ in moving through nodal displacements, **v**, is equal to the work done by distributed loading **F** and Φ, when moving through the displacement field $\mathbf{u} = \mathbf{N}\mathbf{v}$. This is shown with the following equations:

$$W = \mathbf{v}^T \mathbf{S}^0 = -\int_V \mathbf{u}^T \mathbf{F} dV - \int_{S_\Phi} \mathbf{u}^T \Phi dS \tag{2.9}$$

$$\mathbf{S}^0 = -\int_V \mathbf{N}^T \mathbf{F} dV - \int_{S_\Phi} \mathbf{N}^T \Phi dS = \mathbf{S}_F^0 + \mathbf{S}_\Phi^0 \tag{2.10}$$

Where $\mathbf{S}_F^0$ is the nodal load contribution from body forces, and $\mathbf{S}_\Phi^0$ is the nodal load contribution from surface traction.

Lumping of the distributed loads should only be done when working with linear elements with corner nodes. For higher-order shape functions, it is recommended to use consistent nodal loads.

2.1.3 Isoparametric mapping

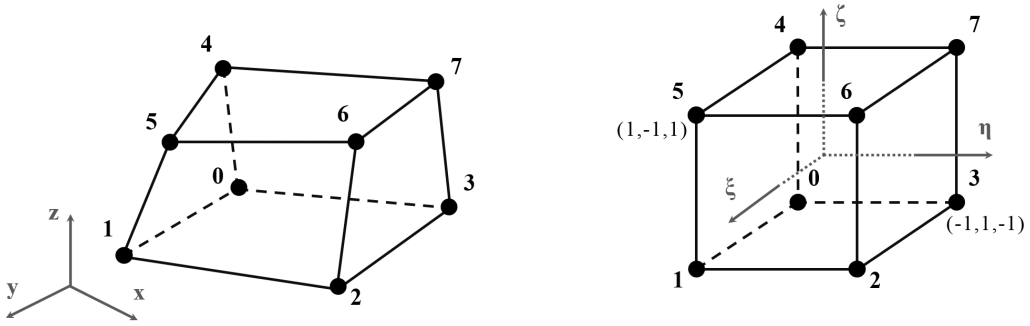


Figure 2.1: Isoparametric mapping of an eight-node hexahedron element.

To account for elements with an arbitrary shape, the element can be formulated as an isoparametric element with natural coordinates. For a hexahedron element, the mapping transforms the element into a quadratic cube with lengths of 2 and the origin in the middle, as shown in Figure 2.1. For the following example of an implementation of the FEM, this type of element is used.

After mapping, the element's shape functions can be expressed with natural coordinates. These consist of eight trilinear functions given in Eq. 2.11.

$$\mathbf{N}_i = \frac{1}{8}(1+\xi\xi_i)(1+\eta\eta_i)(1+\zeta\zeta_i) \tag{2.11}$$

Here, ξ_i, η_i, and ζ_i represent the natural coordinates of node i.

By interpolating the shape functions, $\mathbf{N}$, and the nodal displacements, $\mathbf{r}_e$, the displacement vector, $\mathbf{u}$ is created.

$$\mathbf{u} = \mathbf{N}\mathbf{r}_e, \qquad \mathbf{r}_e = \begin{bmatrix} \mathbf{r}_{e1} \\ \mathbf{r}_{e2} \\ \vdots \\ \mathbf{r}_{e8} \end{bmatrix}, \qquad \mathbf{N} = \begin{bmatrix} \mathbf{N}_1 & \mathbf{N}_2 & \cdots & \mathbf{N}_8 \end{bmatrix} \tag{2.12}$$

Here, the displacements for the different directions, u_i, v_i and w_i, and the shape function are given by Eq. 2.13:

$$\mathbf{r}_{ei} = \begin{bmatrix} u_i \\ v_i \\ w_i \end{bmatrix}, \qquad \mathbf{N}_i = \begin{bmatrix} N_i & 0 & 0 \\ 0 & N_i & 0 \\ 0 & 0 & N_i \end{bmatrix} \tag{2.13}$$

Furthermore, the strain matrix from the Stiffness Matrix section is defined as

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \cdots & \mathbf{B}_3 \end{bmatrix} \tag{2.14}$$

with $\mathbf{B}_i$ defined as the following matrix:

$$\mathbf{B}_i = \begin{bmatrix} N_{i,x} & 0 & 0 \\ 0 & N_{i,y} & 0 \\ 0 & 0 & N_{i,z} \\ N_{i,y} & N_{i,x} & 0 \\ 0 & N_{i,z} & N_{i,y} \\ N_{i,z} & 0 & N_{i,x} \end{bmatrix} \tag{2.15}$$

Here, the notation $N_{i,x}$ corresponds to the partial derivative of N_i with respect to x. Since shape functions are derived from natural coordinates, the *Jacobian matrix* is used to relate the Cartesian and natural coordinate systems. It may be regarded as a scaling factor multiplied by $d\xi d\eta d\zeta$ to create the physical volume $dxdydz$. The Jacobian matrix is derived from the partial derivatives of the shape functions.

$$\begin{bmatrix} N_{i,x} \\ N_{i,y} \\ N_{i,z} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} N_{i,\xi} \\ N_{i,\eta} \\ N_{i,\zeta} \end{bmatrix} \tag{2.16}$$

In Eq. 2.16 the matrix **J** is given by:

$$\mathbf{J} = \begin{bmatrix} N_{i,\xi} \\ N_{i,\eta} \\ N_{i,\zeta} \end{bmatrix} \begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} N_{i,\xi} \\ N_{i,\eta} \\ N_{i,\zeta} \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_8 & y_8 & z_8 \end{bmatrix} \tag{2.17}$$

From this, the local stiffness matrix k can be calculated by Eq. 2.18.

$$\mathbf{k} = \int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} \mathbf{B}^T \mathbf{cB} \, |\mathbf{J}| \, d\xi \, d\eta \, d\zeta \tag{2.18}$$

2.1.4 Numerical Integration

Since the integration in Eq. 2.18 is complex, numerical integration schemes are used to simplify the calculations. In FEM, the most common method is *Gauss integration method*, where the integral is replaced with a sum of weighted integrands that are evaluated at predefined *Gauss points*, (Bathe, 2014). The weights and points can be found in Table 2.1. Eq. 2.19, shows the integration for a 3D element with ξ, η and η being the coordinates and w_i, w_j and w_k the weights.

$\pm\xi_i$		w_j
0	$n = 1$	2.000000
$1/\sqrt{3}$	$n = 2$	1.000000
$\sqrt{0.6}$	$n = 3$	$5/9$
0.000000		$8/9$
0.861136	$n = 4$	0.347855
0.339981		0.347855

Table 2.1: Gauss points and weights.

$$I = \int_{-1}^{+1} \int_{-1}^{+1} \int_{-1}^{+1} f(\xi, \eta, \zeta) \, d\xi \, d\eta \, d\zeta \; = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} w_i w_j w_k f(\xi, \eta, \zeta) \tag{2.19}$$

With n sampling points, the Gaussian quadrature will give an exact integration of a polynomial with an order of $2n-1$. For the eight-node hexahedron element, this corresponds to $n = 2$ in all three directions, since shape functions are linear in all three directions. This gives a total of eight sampling points with coordinates $\xi = \eta = \zeta = \pm 1/\sqrt{3}$, and the corresponding weights, also found in Table 2.1, being $w_i = w_j = w_k = 1$.

2.1.5 Stress and Strain

The strain is calculated in Eq. 2.20 by multiplying the strain matrix **B** found in Eq. 2.15 with global displacements, $\mathbf{d}_e$. The strain is calculated individually for each point and is evaluated in Gaussian integration points.

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_x & \varepsilon_y & \varepsilon_z & \gamma_x & \gamma_y & \gamma_z \end{bmatrix}^T = \Delta u = \mathbf{B}\mathbf{d}_e \tag{2.20}$$

Furthermore, stress is found using *Hooke's law*, which gives a linear relationship between stress and strain when using linearly elastic materials. Eq. 2.21 calculates the stress from the strain

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_x & \sigma_y & \sigma_z & \tau_{xy} & \tau_{yz} & \tau_{zx} \end{bmatrix}^T = \mathbf{C}\boldsymbol{\varepsilon} \tag{2.21}$$

where $\mathbf{C}$ is the material matrix and $\boldsymbol{\varepsilon}$ is the strain. The material matrix, $\mathbf{C}$, for a linearly isotropic material is given in Eq. 2.22

$$\mathbf{C} = \begin{bmatrix} C_1 & C_2 & C_2 & 0 & 0 & 0 \\ C_2 & C_1 & C_2 & 0 & 0 & 0 \\ C_2 & C_2 & C_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & G & 0 & 0 \\ 0 & 0 & 0 & 0 & G & 0 \\ 0 & 0 & 0 & 0 & 0 & G \end{bmatrix}, \qquad \begin{aligned} C_1 &= \tfrac{E}{(1+\nu)(1-2\nu)}(1-\nu) \\ C_2 &= \tfrac{E}{(1+\nu)(1-2\nu)}\nu \\ G &= \tfrac{E}{2(1+\nu)} \end{aligned} \tag{2.22}$$

Since stress and strain have been calculated in the Gauss points, an extrapolation has to be performed to obtain stress and strain in the nodal points. This is done by introducing a new set of coordinates which corresponds to the shape functions being equal to one in the Gauss points.

$$\xi = \xi'/\sqrt{3}, \quad \eta = \eta'/\sqrt{3}, \quad \zeta = \zeta'/\sqrt{3} \tag{2.23}$$

The nodal extrapolated stresses are calculated in Eq. 2.24

$$\sigma_{extr} = \sum_{i=1}^{n_d} N_i' \sigma_i' \tag{2.24}$$

where σ_i' corresponds to the Gauss stress at points i, and N_i' corresponds to the shape function; see Eq. 2.11, of node i, using the new coordinates ξ', η' and ζ'. σ_{extr} is the extrapolated stress.

When working with metallic materials, the most common yield criterion is the *Von Mises* yield criterion, shown in Eq. 2.25. This stress is compared to the yield stress of the material to evaluate utilization.

$$\sigma_m = \sqrt{\frac{(\sigma_x - \sigma_y)^2 + (\sigma_y - \sigma_z)^2 + (\sigma_z - \sigma_x)^2 + 6(\tau_{xy}^2 + \tau_{yz}^2 + \tau_{zx}^2)}{2}} \tag{2.25}$$

2.1.6 Higher-order Shape Functions

The example for *Isoparametric mapping* in 2.1.3, uses an eight-node hexahedral element. It is also possible to use elements with a larger number of nodes. Increasing the number of nodes for each element increases the accuracy and convergence rate, but at a higher computational cost. Since the eight-node element only has nodes in the corner, it is restricted to represent linear shape functions between nodes. This means it cannot represent beam action properly, since its element edges remain straight when the element is bent. On the other hand, the 20-node element is capable of representing quadratic shape functions because of the mid-side nodes, which makes it better to describe beam action. Furthermore, the eight-node element exhibit shear-locking, while the 20-node does not. This makes it possible to converge towards a solution with fewer elements. However, the 20-node element has 60 DOFs per element, in contrast to the eight-node element, which has 24. This, in turn, results in a much larger stiffness matrix and a slower computational time. For the following explanation, the 20-node serendipity element is used.

For higher-order shape functions, the implementation is equivalent to that of the shape function part of Section 2.1.2. The only difference is the functions for the different nodes, (Zienkiewicz & Taylor, 2000). For the corner nodes, the functions are the same as those given in Eq. 2.11. The functions for the mid-side nodes are given by the following equations:

$$\mathbf{N}_i = \frac{1}{4}(1-\xi^2)(1+\eta\eta_i)(1+\zeta\zeta_i) \qquad \text{for } i = 8,10,12,14 \tag{2.26}$$

$$\mathbf{N}_i = \frac{1}{4}(1-\eta^2)(1+\xi\xi_i)(1+\zeta\zeta_i) \qquad \text{for } i = 9,11,13,15 \tag{2.27}$$

$$\mathbf{N}_i = \frac{1}{4}(1-\zeta^2)(1+\xi\xi_i)(1+\eta\eta_i) \qquad \text{for } i = 16...19 \tag{2.28}$$

Here, the nodal numbering is given in Figure 2.2.

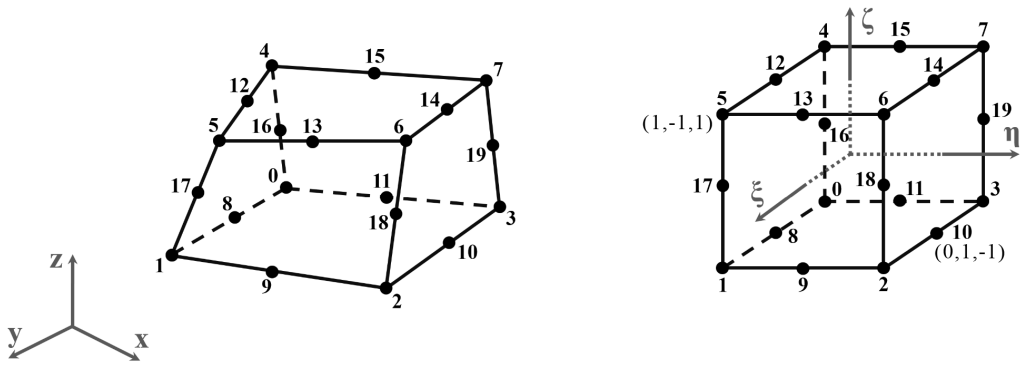


Figure 2.2: Isoparametric mapping of a twenty-node hexahedron element.

Further calculations follow the same principles as those in the previous sections of this chapter. The only difference is an increase in the size of a lot of the matrices. There is also a difference in numerical integration. Full integration of the 20-node element corresponds to $n = 3$ in all three directions. This gives a total of 27 sampling points; see Table 2.1 for coordinates and weights.

2.2 Machine Learning

As can be seen in Figure 2.3 ML is a subcategory under the umbrella term AI. An ML algorithm is an algorithm that is capable of learning from data. It is often used in tasks that are too complex or difficult to solve using human-written programs. Instead of implementing an algorithm with fixed rules, the algorithm can learn from the data provided in training (Goodfellow et al., 2016). To clarify the meaning of learning in this context, the following definition is included:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." Mitchell, 1997

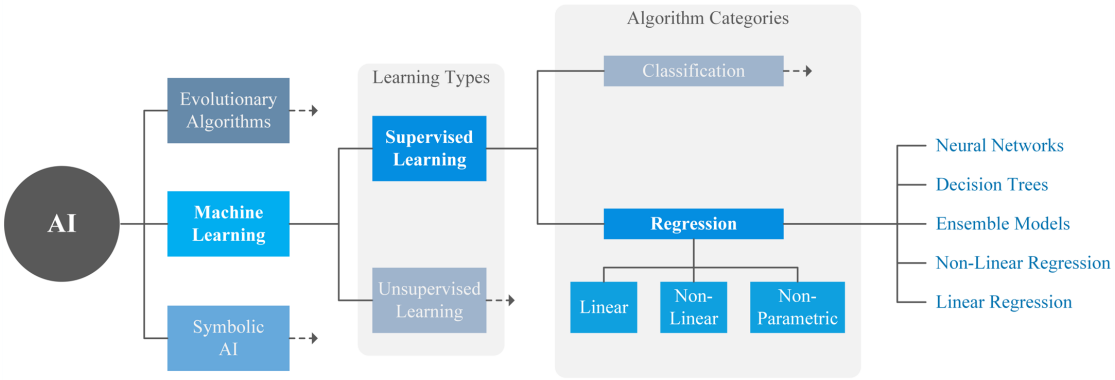


Figure 2.3: Machine learning in the AI landscape.

There are many different classes of tasks that can be solved by machine learning. In this thesis, the tasks to be solved are called regression tasks and are a subcategory of supervised learning. The task is usually described and categorized in terms of how the machine learning system should process an example. One set or example is a collection of features that are usually represented by a vector $\mathbf{x} \in \mathbb{R}^n$ where each x_i is a feature. In the case of supervised learning tasks, each input feature is associated with a label or target. The program tries to learn a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, which maps the features to the labels. Similarly to features, the labels can be represented by a vector $\mathbf{y}$ where each y_j is a label. The experience of a supervised learning algorithm involves observing several samples of random features and labels taken from a larger collection of samples, called datasets. The algorithm then attempts to predict the target from the feature by maximizing the function $p(\mathbf{y}|\mathbf{x})$. The term supervised learning comes from providing both the feature and the label and, in this way, showing the system what to do.

To be able to evaluate the capabilities of the algorithm, a measure of its performance must be introduced. The goal is to minimize the error between the prediction and the target, calculated by a given loss function. There are many loss functions that can be used; the choice depends on the problem at hand and the desired behavior of the algorithm. Sometimes a loss function that penalized frequently made medium-sized mistakes is needed, and other times large mistakes that are rarely made should be penalized hard. It is also possible to create a custom loss function, as will be shown in Case Study 3. In short, a loss function is any function that calculates the degree to which the prediction made by the model is incorrect. Two of the most commonly used functions are called the mean absolute error and the mean squared error. These functions are called mean

because they calculate the loss over an entire dataset and return the mean value. Equation 2.29 illustrates these two loss functions.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - f(x_i)|, \qquad MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(x_i))^2 \tag{2.29}$$

The interesting aspect to observe during training is how the algorithm performs on data that it has never seen before. Therefore, the data are divided into three sets, training, validation, and testing. The training set is used to train the model, the validation set is used to evaluate performance during training, and the testing set is applied to the model after training is completed to get a final evaluation of the model's accuracy. The training and validation loss gives a good indication if the model is overfitting or underfitting. If the model performs very well on the training data but cannot get the same performance on the validation data, it usually means that the model is overfitting. Overfitting is a term that is used when a model learns the detail and noise in the training data so well that it negatively impacts the performance of the model on new data. Random properties from the training data are picked up and learned as concepts by the model. The problem is that new data does not have these concepts and therefore the concepts have a negative impact on the prediction of the model. On the other hand, if performance is low on both the training and validation data, the model is said to underfit. Underfitting is the case where the model can not learn anything from the training data. Ideally, we want a model that performs well both on the training data and on the validation data. Figure 2.4 illustrates underfitting, robust fit and overfitting.

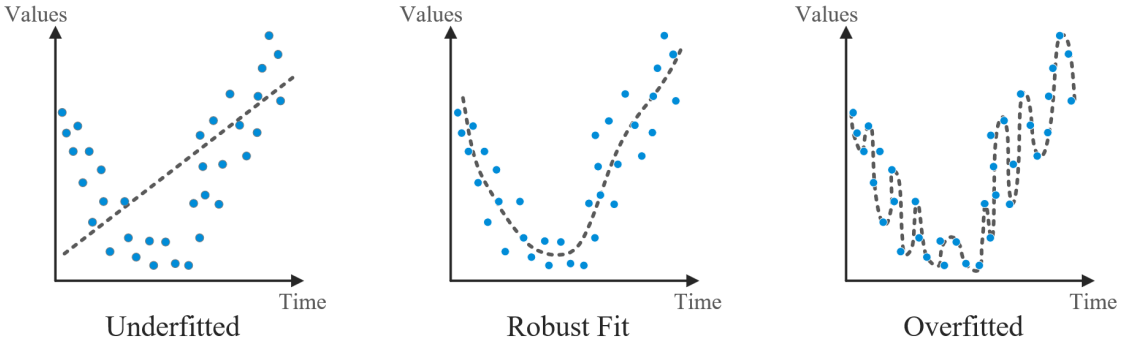


Figure 2.4: Underfitting, robust fit and overfitting of a model.

2.2.1 Linear Regression

A regression task is a task in which the program is asked to predict a numerical value given some input. The simplest case of a regression task is called linear regression. In this case, the output is a linear function of the input. To clarify the notation in Equation 2.30, $\hat{y}$ is the predicted value of the label y, $\mathbf{x}$ is the input vector of features, and $\mathbf{w} \in \mathbb{R}^n$ is a vector of weights. The equation can be written as follows:

$$\hat{y} = \mathbf{w}^T \mathbf{x} \tag{2.30}$$

Equation 2.30 defines the task of predicting y from **x**. In the equation, w_i is multiplied by x_i and summed to obtain the prediction. In this sense, it is intuitive to see how the weights affect the prediction. If a feature's weight is large, it has a large effect on the prediction, and if it is small, it has a small effect. A negative weight reduces the prediction, while a positive weight increases the prediction. In training, these weights start as random values, which often gives a large loss. The weights are then adjusted in a way that reduces the loss provided by the loss function and improves the performance with experience. The goal is to find the weights that minimize the error at all points. This is achieved when the partial derivatives of the loss function with respect to the weights are zero. The optimization algorithm used for this process is called gradient descent, and it involves estimating the gradient for every point, taking a step in the opposite direction of the gradient, and repeating this process until the loss converges towards a minimum. To illustrate this calculation, the normal equations of the mean squared error are shown below. Equation 2.31 illustrates the derivative with respect to the weights of the mean squared error. In this equation, the prediction has the notation $\hat{y}^{(train)}$ and is a result of the input features $\mathbf{X}^{(train)}$ multiplied by the weights **w**. The equation is then solved for the weights.

$$\nabla_w \frac{1}{m} ||\hat{y}^{(train)} - \mathbf{y}^{(train)}||_2^2 = 0 \tag{2.31}$$

$$\implies \nabla_w \frac{1}{m} ||\mathbf{X}^{(train)}\mathbf{w} - \mathbf{y}^{(train)}||_2^2 = 0 \tag{2.32}$$

$$\implies \nabla_w (\mathbf{X}^{(train)}\mathbf{w} - \mathbf{y}^{(train)})^T (\mathbf{X}^{train}\mathbf{w} - \mathbf{y}^{(train)}) = 0 \tag{2.33}$$

$$\implies \mathbf{w} = (\mathbf{X}^{(train)^T}\mathbf{X}^{(train)})^{-1}\mathbf{X}^{(train)^T}\mathbf{y}^{(train)} \tag{2.34}$$

Finally, an additional parameter called bias is included in the equation. The bias b accounts for when the regression line does not include the origin, and thus a translation b is needed. This bias is included in Equation 2.35.

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b \tag{2.35}$$

In this thesis, most regression tasks include problems with multiple inputs and outputs, called multiple regression tasks. These regression tasks work in a similar fashion as described above; in matrix form the equation can be written as illustrated in Equation 2.36.

$$\begin{bmatrix} y_1 & y_2 & \cdots & y_p \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & \cdots & x_q \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1p} \\ w_{21} & w_{22} & \cdots & w_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ w_{q1} & w_{q2} & \cdots & w_{qp} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & \cdots & b_p \end{bmatrix} \tag{2.36}$$

2.2.2 Feedforward Neural Networks

The goal of feedforward networks, or deep feedforward networks if the depth of the network is two or greater, is to approximate a function. The network defines the mapping $\mathbf{y} = f(\mathbf{x};\theta)$ and learns the parameters θ that give the best approximation. It is called feedforward because the information flows through the function evaluated from $\mathbf{x}$, through the computations that define the function, and finally to the output. In other words, there are no feedback connections that feed the output back into itself, as in recurrent neural networks. This is an important distinction, because in a feedforward network, the gradient is clearly defined and computable through backpropagation or the chain rule. Feedforward networks form the basis of many important networks, such as fully connected networks and convolutional networks, which we will describe later in this section. The reason it is called a network is because it can consist of many different functions connected in a chain or a network. As an example, a network with three hidden layers has three functions connected in a chain, $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. Figure 2.5 illustrates this network as a fully connected network with all features $\mathbf{x}$ in the input layer, $f^{(1)}$ as the first hidden layer, $f^{(2)}$ as the second hidden layer, $f^{(3)}$ representing the third hidden layer, and the last layer is called the output layer, which consists of all the labels.

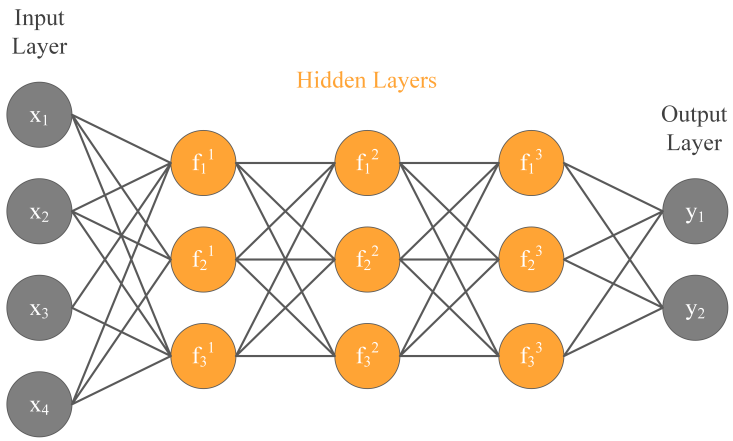


Figure 2.5: Illustration of a fully connected feedforward network with three hidden layers.

The length of the chain represents the depth of the model and is the reason for the terminology "deep learning". In the other direction, the number of neurons in the layers determines what we call the width of the model. Each feature $\mathbf{x}$ is mapped to a label $\mathbf{y}$ and during training the output layer is told what to do at each point. It must produce a value as close to the label as possible. The other layer is not told directly what to do, but the learning algorithm must decide how to use these layers to obtain the best approximation of the function. Because these layers are not told directly what to do, they are called hidden layers.

Each neuron in the network is a function that receives a value from the connected neurons. A weighted sum of all these inputs, with an added bias value, is calculated, and we obtain the activation of the neuron, $a = g(\mathbf{w}^T\mathbf{x} + b)$. By applying a non-linear activation function on this calculated value, we have introduced non-linearity to the model. When including this non-linear activation function, we get a non-linear regression task that outputs a non-linear function of the input. Without this activation function, we would not be able to approximate any continuous func-

tion; therefore, it is very important for nonlinear problems. The most common activation function is called ReLU, short for rectified linear unit.

$$R(x) = \max(0, x) \tag{2.37}$$

ReLu returns x for all positive values, zero for negative values. The function is very simple; there is no complicated math or heavy computations. Due to this simplicity, the model often takes less time to train and run compared to other activation functions. This is one of the reasons why ReLU has been shown to learn faster than other activation functions, (Team, 2020).

The optimization algorithm used in this thesis is called the Adam optimizer and is an optimization technique for gradient descent. Because this algorithm requires less memory, it is efficient when working on large problems involving a lot of data. The optimizer inherits the strengths of two gradient descent methods, "gradient descent with momentum" and "root mean square propagation" (RMSP), to obtain a more optimized gradient descent method that reach the global minimum efficiently, (Prakhar, 2020). Next, some of the network architectures explored during development are presented.

Fully connected neural network

A fully connected network consists of a series of fully connected layers that connect every neuron in one layer to every neuron in the following layer. Figure 2.5 is an example of a fully connected neural network. A fully connected network has the simplest architecture and is very broadly applicable. On the other hand, it tends to have a weaker performance than special-purpose networks designed for the problem at hand, (Bharath Ramsundar, 2018).

Convolutional neural network

As the fully connected network, the convolutional network is made up of neurons, each neuron receives input, performs a dot product, and follows with nonlinearity. There are three types of layers that are used to build the convolutional architecture, the convolutional layer, the pooling layer, and a fully connected layer as the output layer. The difference is that the convolutional architecture makes the assumption that the inputs are images, which allows us to encode certain properties into the architecture. These properties make the forward function more efficient to implement and reduce the number of parameters in the network. In comparison the fully connected network does not scale well to full images. For example, an image with low resolution, such as 32x32 pixels with 3 channels, seems manageable, but once the image resolution gets larger, the network struggles. An image with resolution 512x512 would lead to an input vector that has 512x512x3 = 786 432 features. Unlike a fully connected network, a convolutional network takes advantage of the fact that the input is an image and arranges the features in three dimensions: width, height, and depth. Depth in this context refers to the third dimension, not the number of layers, as described earlier in feedforward networks. An image, 512x512x3, has a width and height of 512 and a depth of 3. To clarify, an example architecture that includes all the layers used to build a convolutional network is illustrated in Figure 2.6.

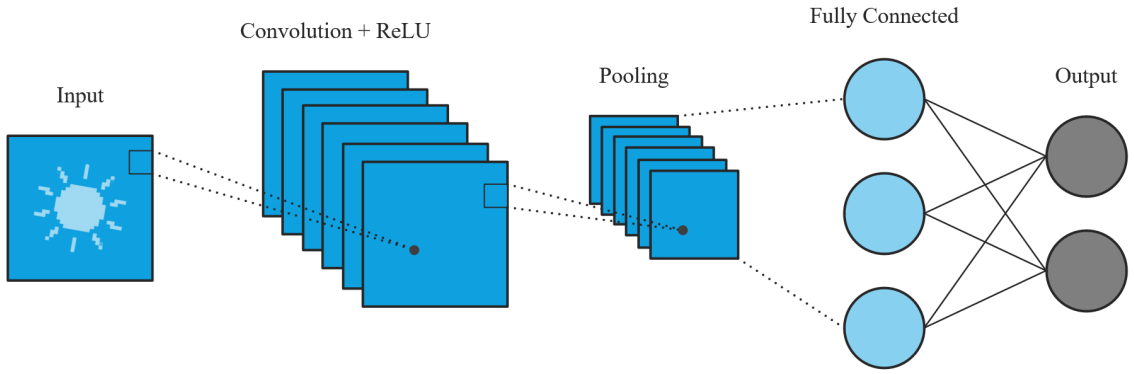


Figure 2.6: Illustration of a convolutional network architecture.

Steps in a simple convolutional network:

- Input: An image [32,32,3] has a width and height of 32 with the three channels R,G,B.
- Convolutional layer: Compute the output of neurons that are connected to local regions in the input. Using six filters, the result is a volume with six feature channels [32,32,6].
- ReLU layer: Introduces nonlinearity and does not change the dimensions.
- Pool layer: Performs a downsampling operation in spatial dimensions (width and height), resulting in a volume of [16,16,6].
- Fully connected layer: In this case, the output layer depends on the desired output of the network. For example, [1,1,2].

Note that the sequence is not fixed and that all layers must not be included in every sequence. For example, another network can look like this: [Input-Conv-ReLU-Conv-ReLU-Pool-Conv-ReLU–Conv-ReLU-Pool-FC] (Stanford, 2022). Also, it is worth mentioning that in our research we have a grid of points that can be associated with the pixels, and instead of RGB values for each pixel or point, we use a scalar value or a vector. In these cases, we get the dimensions [32,32,1] or [32,32,3] if the dimension of the grid is 32.

Traditional Unet

Unet is a fully convolutional network, very popular in image segmentation. The architecture is often used to transform from one input image to another output image. Unet is a U-shaped encoder-decoder network architecture, which usually consists of four encoder blocks and four decoder blocks that are connected with a bridge at the bottom. The architecture is illustrated in Figure 2.7. At each encoder block, the spatial dimensions (width and height) are reduced by a factor of two, while the number of feature channels is doubled. In the same way, the decoder network doubles the spatial dimensions and halves the number of feature channels.

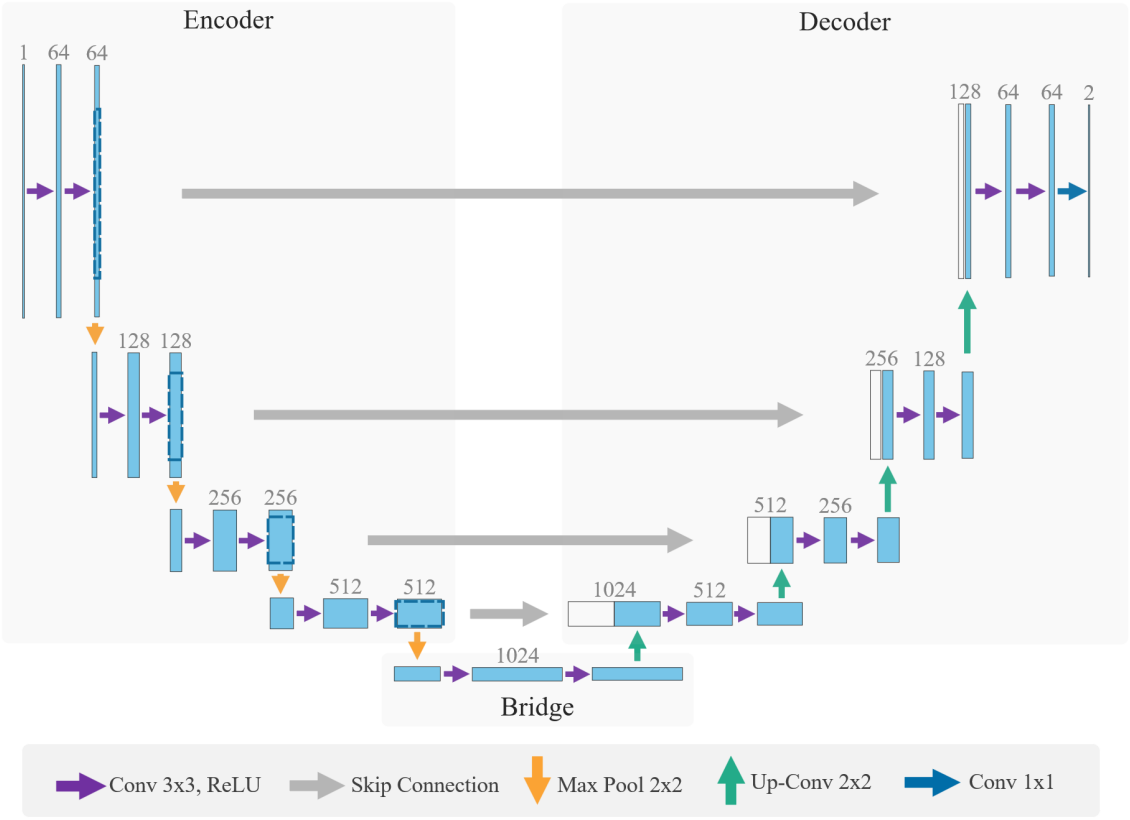


Figure 2.7: Illustration of the Unet architecture.

The encoder block consists of two 3x3 convolutions with ReLU that introduce nonlinearity into the model. In Figure 2.7, these convolutions are marked with a purple arrow. The output from the last ReLU function is sent to the corresponding decoder block; this operation is called a skip connection, and is visible in the figure as gray arrows. In a sense, the encoder block acts as a feature extractor for the decoder block and provides the decoder with additional information that helps to improve the flow of gradients during backpropagation. After the convolutions, a 2x2 Max-pooling, shown with orange arrows, is applied to reduce the number of parameters to be trained. As a result of the reduced parameters, the computational cost of the system is decreased.

The bridge at the bottom of the Unet connects the encoder to the decoder network. The bridge consists of two 3x3 convolutions with ReLU functions in between.

The decoder network receives the feature representation provided by the encoder block through the skip connection and generates a segmentation mask. The decoder block contains a 2x2 transpose convolution indicated with green arrows in Figure 2.7. The result of this convolution is concatenated with the information provided by the skip connection, and thereafter the result is sent into two 3x3 convolutions with ReLU in between the convolutions. At the end of the network, the output is passed through a 1x1 convolution followed by a sigmoid function. This gives a segmentation mask that represents the pixel-wise classification. Classification is the process of predicting the class of given data points. For example, in the case of image segmentation and classification, an image of a dog can have the classes dog, grass, sky, and tree. The task in this example is to identify to which class a given data point belongs, (Stanford, 2022).

2.3 Rotational Stiffness in Structural Connections

2.3.1 Introduction

Connections are an integral part of a structure and require special attention to ensure safe and cost-effective construction. The way forces are absorbed in a structural element and transferred to the next one, and how the structural system works as a whole, is highly dependent on the types of joints and connections. The connections used to join the structural elements can be in the form of a point, a line, or a surface. A point connector is usually called a pinned joint, whereas line and surface connectors are termed rigid or fixed joints. These terms of connections are idealizations of actual physical joints, where their behavior can differ greatly from reality. A pinned joint, e.g. a bolted connection, allows rotation but resists translation in any direction. A rigid or fixed joint, for example, a welded connection or a glued surface, can also restrain rotation and provide both force and moment resistance.

2.3.2 Classification of joints

The properties of a joint can be determined by considering its rotational and translational behavior under loading. There are different possible ways to classify connections, and three widely used experimental approaches are illustrated in Figure 2.8.

- **Strength** - the moment resistance. The connection can be full-strength, partial-strength, or nominally pinned.
- **Rigidity** - the rotational stiffness. The connection may be rigid, semi-rigid, or nominally pinned.
- **Ductility** - the rotation capacity of the joint before it collapses. The connections can be classified as infinitely ductile, limited ductile, and non-ductile.

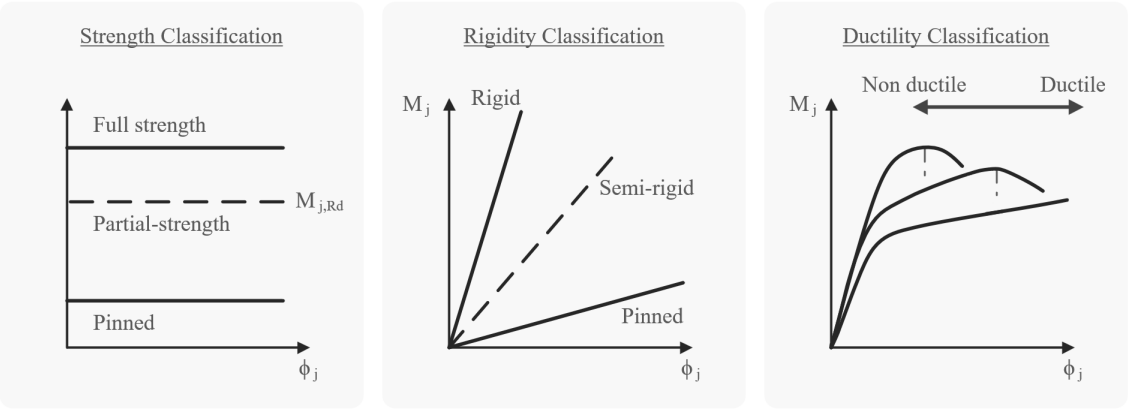


Figure 2.8: Three different joint classification approaches. (*Joints in Steel Construction Moment Connections*, 1995)

2.3.3 Rotational Stiffness

In this thesis, the rigidity classification method will be studied. For simplicity, an elastic response is considered on a beam-to-column connection. The definition of rotational stiffness, S_j, is defined by Equation 2.38.

$$S_j = \frac{M_j}{\phi_j} \tag{2.38}$$

where M_j is the bending moment in the connection and ϕ_j is the difference between the absolute rotations of the two connected members. When the rotational stiffness, S_j, is zero or relatively small, the joint can be classified as pinned. On the contrary, when the rotational stiffness is infinite or relatively high, the joint will fall into the rigid joint class. A pinned or rigid joint is an idealized assumption that is used in structural calculations, and in reality, no joint will have zero or infinite stiffness. For all intermediate cases, the joint will belong to the semi-rigid joint class. Taking into account an elastic response, the relationship of the moment and the rotation is linear, and the rotational stiffness will be the gradient of the $M-\phi$ line. This is called the initial rotational stiffness with the symbol $S_{j,ini}$ (ECCS, 2016).

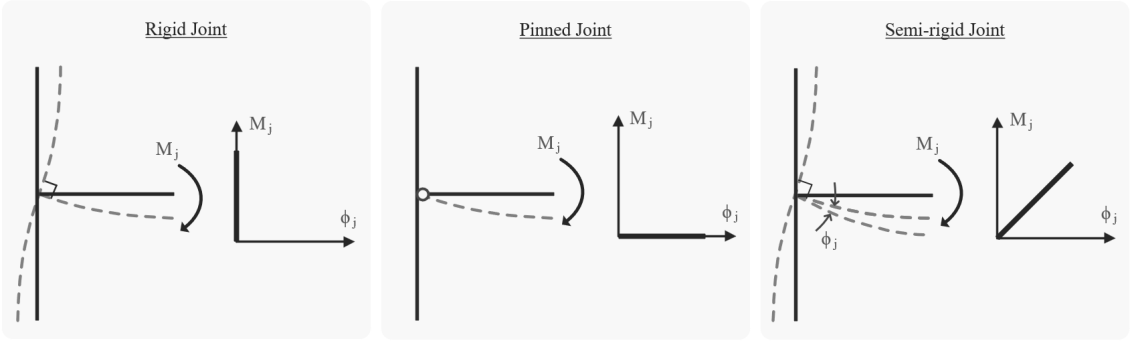


Figure 2.9: Classification of joints according to stiffness and their moment-rotation diagram (*Joints in Steel Construction Moment Connections*, 1995).

There are many ways to analyze and compute the rigidity of a semi-rigid joint. A simplified numerical method using FEM software will be introduced, and the approach according to Eurocode 3 will be briefly explained.

Numerical method using FEM software

This is a simplified method, but will give an approximation of the rotational stiffness of a beam-to-column connection subjected to a bending moment, M_j. If ϕ_b and ϕ_c are the rotation of the beam and column, respectively, the rotation of the connection, ϕ_j, is defined in Equation 2.39.

$$\phi_j = \phi_b - \phi_c \tag{2.39}$$

To find the angle of rotation of the beam, an evaluation point, p_b, can be placed on the beam axis a distance x_{pb} from the connection point. D_{z1} will be the vertical displacement of the intersection

point of the beams, i_b. D_{z2} will be the vertical displacement of p_b. See Figure 2.10. The angle of rotation of the beam can then be calculated using the following formula.

$$\phi_b = arctan\left(\frac{D_{z2} - D_{z1}}{x_{pb}}\right) = arctan\left(\frac{D_z}{x_{pb}}\right) \approx \frac{D_z}{x_{pb}} \tag{2.40}$$

A similar approach can be used to find the angle of rotation of the column. Where the evaluation points, p_{c1} and p_{c2}, can be placed on the vertical column axis, with an equal length of $\frac{1}{2}z_{pc}$ from the intersection point of the columns, i_c. z_{pc} is the lever arm and can be defined as $z_{pc} = h_b - t_{fb}$, where h_b is the total height of the beam and t_{fb} is the thickness of the beam flanges. The difference in the horizontal displacement of p_{c1} and p_{c1} will be D_x, and the angle can be obtained from:

$$\phi_c = arctan\left(\frac{D_{x2} - D_{x1}}{z_{pc}}\right) = arctan\left(\frac{D_x}{z_{pc}}\right) \approx \frac{D_x}{z_{pc}} \tag{2.41}$$

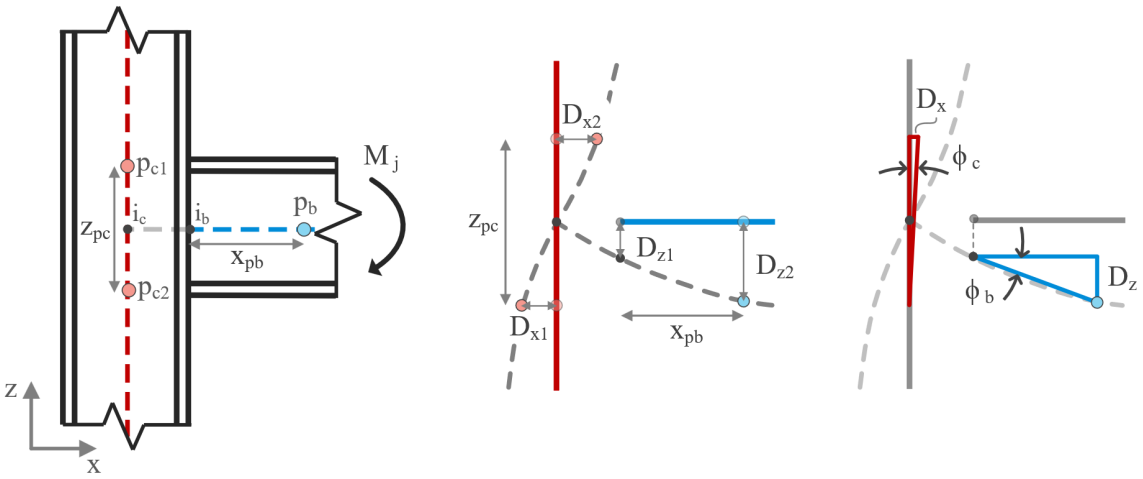


Figure 2.10: Numerical computation of rotational stiffness of a beam-to-column connection.

With the applied bending moment, M_j, the initial rotational stiffness, $S_{j,ini}$, of the beam-to-column connection can be calculated by equation (2.38).

Empirical method - Eurocode

In *Eurocode 3: Design of steel structures - Part 1-8: Design of joints*, 2009 a guide is provided to assess rotational stiffness for many types of joints. This practical application is limited by criteria that are not always verifiable. For example, the general Equation (2.42) is only applicable if the design value of the axial force, N_{Ed}, in the connected member is greater than 5% of its plastic resistance in the cross section. Additionally, the equation is also only valid for joints connecting I or H sections. If these conditions are satisfied, the following formula for rotational stiffness from Equation 6.27 in EN 1993-1-8 can be applied.

$$S_j = \frac{S_{j,ini}}{\mu} = \frac{Ez^2}{\mu \sum_i \frac{1}{k_i}} \tag{2.42}$$

where E is the Young's Modulus for the material of the connection, k_i is the stiffness coefficient for the various i components and z is the lever arm which will change with the type of joint. The stiffness ratio, μ, can be set as 1 if the design bending moment, $M_{j,Ed}$, is less than 2/3 of the moment resistance of the connection, $M_{j,Rd}$. So, for an elastic analysis, μ is 1 and $S_{j,ini}$ is given by the same expression as for S_j.

3 Software

This section describes relevant software for this thesis. The software has been classified into AAD software, FEM software, and Programming software, where AAD software has been used mainly to create geometry. FEM software has been used to perform FEA. Finally, the programming software has been used to develop plugins for the Grasshopper and ML algorithms.

3.1 AAD Software

3.1.1 Rhino/Grasshopper (version 7)

Rhinoceros 3D is a CAD software, (McNeel, 2022). The software makes it possible to create complex 3D geometry. Grasshopper is a Rhino plugin that allows AAD. Grasshopper provides a visual programming tool to create geometry in Rhino. This differs from traditional programming by allowing for a more graphical approach to programming. Instead of writing lines of code, different components are connected with lines to create code. This means that an AAD model can be automatically updated when its input changes. By creating geometry with parameters on sliders, different design proposals can be created in a short amount of time. Figure 3.1 shows a simple example of creating geometry with Grasshopper.

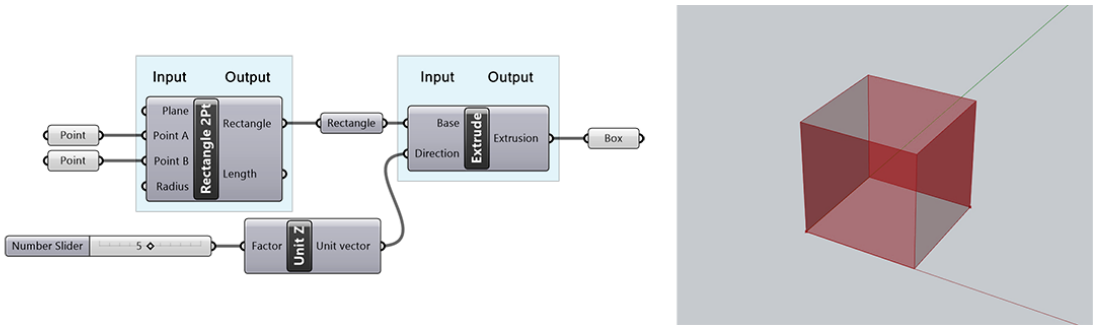


Figure 3.1: Example of parametric modelling in Rhinoceros 7 with Grasshopper

3.2 FEM Software

3.2.1 Karamba3D

Karamba3D is an interactive, parametric Finite Element program, (Preisinger, 2013). Karamba3D is a plugin for Grasshopper. This makes it easy to combine parameterized geometric models and finite element calculations. It has tools to perform structural analysis by transforming geometry into finite element objects. The limitation of Karamba3D is that it does not support 3D elements, only beam and surface elements.

3.2.2 Abaqus

Abaqus is a FEM software with extensive possibilities, (Company, n.d.). It can perform simple linear and more complex nonlinear analyses. The software supports a wide range of elements and materials. One of the strengths of Abaqus is the ability to customize the model and problem with a high level of detail. Abaqus is one of the leading FEM software in the civil engineering world, especially when it comes to solid element problems. This is the reason why it is used for verification throughout this thesis.

3.2.3 Robot Structural Analysis

Robot Structural Analysis is a FEM software by Autodesk, (Autodesk, n.d.). It is a simpler and more graphical software than Abaqus. The analyzes are mostly based on beam and shell elements. For this thesis, it has been used to calculate the rotational stiffness and compare with the calculations performed in Grasshopper. Robot uses the method for rotational stiffness analysis in the Eurocodes directly.

3.3 Programming Software

3.3.1 Visual Studio 2022

Visual studio 2022, (Microsoft, 2022a), has been used to create plug-ins for Grasshopper. Codes have been written in C#. *MathNet*, ('Math.NET Numerics', n.d.), has been used for mathematical and matrix operations, and *CSparse*, ('CSparse', n.d.), for more efficient matrix calculations. This framework is optimal for creating plug-ins for Grasshopper, since it has templates for Rhino 7 and Grasshopper. Grasshopper is developed in C#, making it easy to use its custom API when creating new plug-ins.

3.3.2 Visual Studio Code

Visual Studio Code has been used to program the ML, (Microsoft, 2022b). These codes were written in Python. This framework has been used since it has better debugging features than Visual Studio 2022.

Packages

PyTorch is an open-source ML framework. It is an optimized tensor library for deep learning that uses GPUs and CPUs. The networks have been built using PyTorch. *NumPy* (Harris et al., 2020) has been used for computational tasks, and *MatplotLib* (Hunter, 2007) for plotting. In addition, *Pygmsh* (Schlömer, 2020) and *PyVista* (Sullivan & Kaszynski, 2019) has been used for mesh generation. For visualizing results and collaboration, *WandB* (Biewald, 2020) has been used, which is a platform designed to support and automate key steps in the development of ML models.

4 Methods

4.1 Simple FEM Solver Plugin with 8-node Hex Element

In this first plugin, a system analysis of a solid problem is created using a trilinear hexahedron, often called the 8-node brick element, for the grasshopper environment. The plugin is divided into three main components, **loft mesh**, **FEM solver**, and **preview results**. **Loft mesh** is a preprocessing component in which a mesh with trilinear hexahedron elements is created with the desired density. The **FEM solver** component is applied to calculate the nodal displacements and to calculate the strains and stresses. Finally, the **Preview** components are created to show the results in the most optimal way.

Before discussing the study in detail and for clarification, Table 4.1 presents the custom classes used in the plugin.

Class name	Properties	Description
Element	id, Nodes, Mesh	An element has an id, eight nodes and a mesh.
Node	Global id, Local id, Point3D, Support	A node has a global id in the range zero to the total number of nodes in the system, a local id in the range zero to seven, a point in the 3D space and a support.
Mesh	Faces, vertices	A mesh consist of six faces and eight vertices.
Load	Vector3D, Point3D	A load has a load vector and a point in the 3D space where the load vector is applied.
Support	Point3D, Boolean	A support has a point in 3D space and three Boolean values representing the points ability to move freely in the x-, y- and z-direction.
Material	Young's Modulus, Poisson's ratio	A material has a name, Young's modulus and Poisson's ratio.
Result	Displacements, stresses, strains, new points, old points, mesh	A result has displacements, stresses, strains, new location of the nodes, old location of the nodes and a new mesh.

Table 4.1: Custom classes used in the FEM Solver plug-in.

4.1.1 Loft mesh component

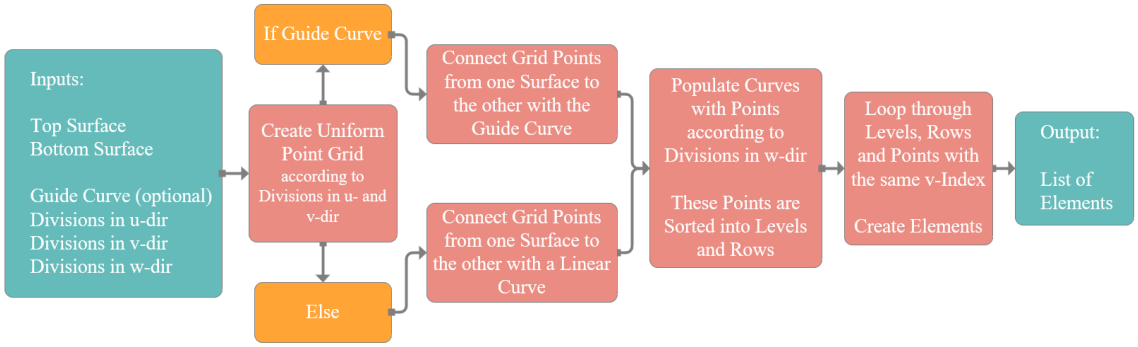


Figure 4.1: Flowchart of the Loft mesh component.

As input, the **Loft mesh** component takes two rectangular surfaces, one optional guiding curve, and divisions in all three directions, u, v, and w. Given these inputs, the component starts by populating the top surfaces with a grid of evenly distributed points according to the division inputs u and v. If a guiding curve is given, this curve is duplicated and used to connect the grid points from one surface to the other. If this guiding curve is not provided, instead a linear curve is created between the points. The curves are then populated with points according to the selected w-division. These points are then sorted into different levels and rows to obtain a data structure that is easy to work with. All points with the same w index belong to the same level, and all points with the same v index belong to the same row. Looping through levels, rows, and points creates the necessary nodes, meshes, and finally elements. These elements are then appended to the output list of this component.

4.1.2 FEM Solver component

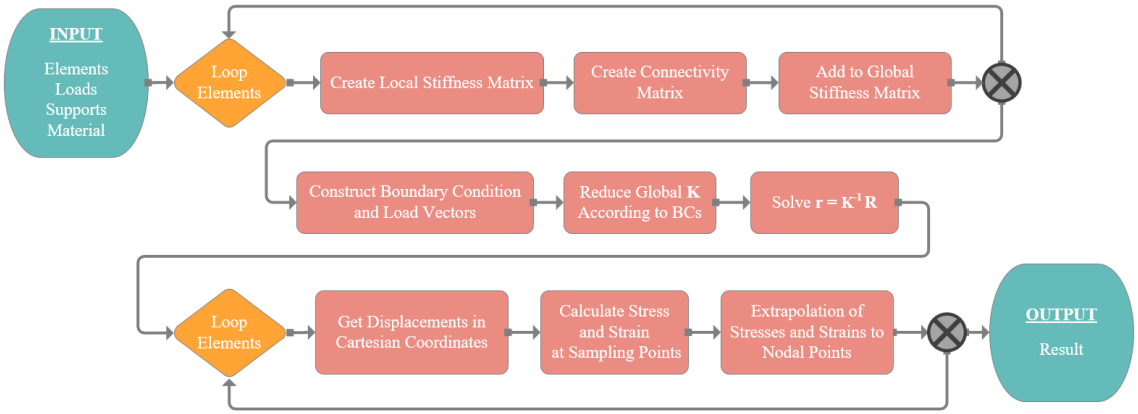


Figure 4.2: Flowchart of the FEM solver component.

The **FEM Solver** component has a total of four different inputs, all provided by custom components. The first input is a list of elements, then a list of loads is needed, a list of supports, and finally a material. To make the main code more slender and increase readability, the **FEM Solver** component uses eight custom methods that are called from the main code. Figure 4.2 shows a

component flowchart. To best describe the solver component, the methods are presented first before going through the main code. Table 4.2 describes some abbreviations used as input in the methods.

Abbreviations	Description in the 8-node brick element case
nodeIdL	Local node id, an integer between zero and seven.
NnodesL	Number of local nodes, in this case eight.
nodes	A list of all nodes in the system.
ndofs	Number of degree of freedom for one node, in this case three.
NnodesG	Number of global nodes in the system.
r	Global nodal point displacements.

Table 4.2: Abbreviations used as input in the methods described.

GetGeneralizedCoordinate (nodeIdL): This method uses the local node id and returns the generalized coordinates for this specific node.

return: $[\xi_i \ \eta_i \ \zeta_i]$.

GetShapeFunctions (NnodesL, ξ, η, ζ): For each of the eight nodes the generalized coordinates are found by calling the **GetGeneralizedCoordinate** method. These coordinates are then used to create a shape function N_i, where i is the node id, and the function depends on the generalized coordinate input values ξ, η and ζ. All shape functions are then appended to the shape functions row vector, $\mathbf{N} = [N0 \ N1 \ N2 \ N3 \ N4 \ N5 \ N6 \ N7]$. Finally, this vector is inserted on the diagonal of a 3 by 3 matrix, where all other elements in the matrix contain a zero row vector of the same length as the **N** vector. In this specific case, the shape function matrix gets dimensions 3 by 24 and is the returned matrix of this method.

return: $\begin{bmatrix} \mathbf{N} & 0 & 0 \\ 0 & \mathbf{N} & 0 \\ 0 & 0 & \mathbf{N} \end{bmatrix}$

GetDerivatedShapeFunctions (NnodesL, ξ, η, ζ): Similar to the **GetShapeFunctions** method the generalized coordinates are found for each node. These coordinates are then used to create three shape functions $N_{i,\xi}$, $N_{i,\eta}$, and $N_{i,\zeta}$, representing the derivative of N_i with respect to ξ, η and ζ. As before, i is the node id and the shape function N_i depends on ξ, η and ζ. The derivative shape functions $N_{i,\xi}$, $N_{i,\eta}$, and $N_{i,\zeta}$, are then, respectively, appended to the row vectors $\mathbf{N}_{,\xi}$, $\mathbf{N}_{,\eta}$, and $\mathbf{N}_{,\zeta}$. The return for this method is a 3 by 8 matrix, where the first row contains the shape functions derivated with respect to ξ, $\mathbf{N}_{,\xi} = [N0_{,\xi} \ N1_{,\xi} \ \dots \ N7_{,\xi}]$. Similarly, the second row contains the $\mathbf{N}_{,\eta}$ vector and the last row is represented by the $\mathbf{N}_{,\zeta}$ vector.

return: $\begin{bmatrix} \mathbf{N}_{,\xi} \\ \mathbf{N}_{,\eta} \\ \mathbf{N}_{,\zeta} \end{bmatrix}$.

ConstructIntegrandForStiffnessMatrix (nodes, material, IntegrationPoint): The goal of this method is to find the integrand $\mathbf{B}^T\mathbf{CB}\,|\mathbf{J}|$ of the stiffness matrix. The first step is to create an 8 by 3 matrix of Cartesian coordinates $\begin{bmatrix}\mathbf{x}\ \mathbf{y}\ \mathbf{z}\end{bmatrix}$, where $\mathbf{x} = \begin{bmatrix}x_1\ x_2\ \cdots\ x_8\end{bmatrix}^T$ and similarly for the y- and z-coordinates. By calling the **GetDerivatedShapeFunctions** for the input integration point and multiplying the return value with the coordinate matrix we get a 3 by 3 Jacobian matrix. To find the 3 by 8 shape function matrix derivated with respect to the Cartesian coordinates, the inverse of the Jacobian matrix is multiplyed with the return value of **GetDerivatedShapeFunctions**. The result of this multiplication is the following matrix, $\begin{bmatrix}\mathbf{N}_{,\mathbf{x}}\ \mathbf{N}_{,\mathbf{y}}\ \mathbf{N}_{,\mathbf{z}}\end{bmatrix}^T$, where $\mathbf{N}_{,x} = \begin{bmatrix}N0_{,x}\ N1_{,x}\ \ldots\ N7_{,x}\end{bmatrix}$ and similarly for the y- and z-derivative $\mathbf{N}_{,y}$ and $\mathbf{N}_{,z}$. Once the derivative with respect to Cartesian coordinates is obtained, the **B**-matrix is constructed by mapping the row vectors into its correct place, resulting in a **B**-matrix with the dimensions 6 by 24. The 6 by 6 **C** matrix is fairly simple to construct when the material properties are given. Finally, the integrand matrix is calculated and returned along with the **B**- and **C**-matrix.

return: $\mathbf{B}^T\mathbf{CB}|\mathbf{J}|$, $\begin{bmatrix}\mathbf{N}_{,\mathbf{x}} & 0 & 0\\ 0 & \mathbf{N}_{,\mathbf{y}} & 0\\ 0 & 0 & \mathbf{N}_{,\mathbf{z}}\\ \mathbf{N}_{,\mathbf{y}} & \mathbf{N}_{,\mathbf{x}} & 0\\ 0 & \mathbf{N}_{,\mathbf{z}} & \mathbf{N}_{,\mathbf{y}}\\ \mathbf{N}_{,\mathbf{z}} & 0 & \mathbf{N}_{,\mathbf{x}}\end{bmatrix}$, $\begin{bmatrix}C_{11} & C_{12} & C_{13} & 0 & 0 & 0\\ C_{21} & C_{22} & C_{23} & 0 & 0 & 0\\ C_{31} & C_{32} & C_{33} & 0 & 0 & 0\\ 0 & 0 & 0 & C_{44} & 0 & 0\\ 0 & 0 & 0 & 0 & C_{55} & 0\\ 0 & 0 & 0 & 0 & 0 & C_{66}\end{bmatrix}$

GetIntegrationPointsBrickElement (): This method returns a list of integration points. In this case, it returns the integration points of an eight-point quadrature rule (2x2x2), corresponding to full integration of the stiffness matrix.

return: $\begin{bmatrix}\xi_0 & \eta_0 & \zeta_0\\ \xi_1 & \eta_1 & \zeta_1\\ \vdots & \vdots & \vdots\\ \xi_7 & \eta_7 & \zeta_7\end{bmatrix}$

ConstructGlobalStiffnessMatrix (elements, ndofs, NnodesG, material): This method starts with creating an empty global stiffness matrix with the correct dimensions. The stiffness matrix is a square matrix with dimensions ndofs times NnodesG, in this case three times NnodesG. Thereafter, **GetIntegrationPointsBrickElement** is called to get the correct integration points for this type of element. The remaining part is to go through the list of elements and, for each element, create the connectivity matrix, the local stiffness matrix, and map the local stiffness matrix onto the global stiffness matrix using the connectivity matrix. The connectivity matrix for each element is easily found by going through the nodes of the current element, extracting the properties local- and global-id for each node, and using this to populate the connectivity matrix with a unit value in the correct places. The local stiffness matrix is, in the case of trilinear hexahedrons, a matrix with dimension 24 by 24. The matrix is then populated by calling the **ConstructIntegrandForStiffnessMatrix** method for each integration point, extracting the integrand matrix, and summing the contributions for each integration point. After finding the connectivity and stiffness matrix for the current element, its contribution, $\mathbf{a}^T\mathbf{ka}$, to the global stiffness matrix is calculated. After going through each element and summing up all the contributions, the global stiffness matrix, K, is ready to be returned.

return: $\begin{bmatrix} K_{11} & K_{12} & \cdots & K_{1n} \\ K_{21} & K_{22} & \cdots & K_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ K_{n1} & K_{n2} & \cdots & K_{nn} \end{bmatrix}$

ConstructSupportsAndLoadVectors (supports, loads, nodes, ndofs): This method takes as input a list of supports and a list of loads. At the start of the method, an empty output list is created for the load vector and support points, both with the same length (ndofs times NnodesG). To create the support output, the method goes through every node in the system and checks if its location corresponds to the location of one of the supports in the input list. If this is the case, the code duplicates the boundary condition for this support and places a Boolean value in the output support list according to its global id. The value is set to True for fixed supports and False for free. The loads are created in a similar fashion; if the location of a node corresponds to the location of one of the input loads, then the load is duplicated and inserted into the output load list at the correct location.

return: $\begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_i \end{bmatrix}, \begin{bmatrix} Bool_0 \\ Bool_1 \\ \vdots \\ Bool_i \end{bmatrix}$

CalculateResults (elements, r, K, material, nodes, ndofs): The method starts by creating a list for each output item: displacements, stresses, strains, and new points. Here, the new points are the locations of the nodes after deformation. The method also needs the integration points for the element used; this is obtained by calling the method **GetIntegrationPointsBrickElements**. The results are created by going through each element of the system. The first operation is to create a connectivity matrix, **a**, for the element; this is done in a similar fashion as described in the **ConstructGlobalStiffnessMatrix** method. Subsequently, the displacements of the local nodal point are calculated from the equation $v = ar$. At this point, the nodal displacements are in the natural coordinate system. To return the displacements to Cartesian coordinates, the **GetGeneralizedCoordinate** method is called for each node of the element. The generalized coordinates are then sent to the **GetShapeFunctions** method to obtain the shape function for the current node. The displacement in Cartesian coordinates can now be found by premultiplying the shape function with the displacement in natural coordinates. The result is a displacement column vector with a length of three, where the first element represents the displacement in the x direction, the second in the y direction, and the last element is the displacement in the z direction. After going through each node in the element, the result is a 3 by 8 displacement matrix where the first column represents the displacement column vector for the first node, the second column for the second node, etc. Each element produces a new displacement matrix that is appended to the output displacement list.

The next task of this method is to find the stresses and strains of each element. This is done by calculating the stresses at the optimal sampling points, in this case the integration points. These stresses are then extrapolated to the nodes. To find these strains and stresses, it is necessary to loop through all integration points. For each integration point, **ConstructIntegrandForStiffnessMatrix** are called. This outputs the **B** and **C** matrices. Multiplying the displacement vector **v** by the

strain matrix **B**, the strains are given. Furthermore, stresses are calculated by multiplying the strain by the material matrix **C**. These strains and stresses are kept in a 6 by 8 matrix, one column for each sampling point, and rows with strain and stress in different directions. To calculate the nodal strain and stress, it is necessary to loop through all nodal points and extrapolate the strains and stresses. This is achieved by first calling **GetGeneralizedCoordinate**, then calling **GetShapeFunctions** with the calculated generalized coordinate multiplied by $\sqrt{3}$. Lastly, the new shape functions are premultiplied with the stress and strain at the sampling points, producing the nodal stress and strain. These stresses and strains are appended to the output list.

The lists of displacements, stresses, strains, and new points are added to the result class and outputted.

return: [ResultClass]

Main code:

The main code begins with obtaining the input, which is a list of elements, list of supports, list of loads and a material. Then a list of all nodes is created without duplicates. This is important, to know how many unique nodes there are, since that determines the size of the stiffness matrix. Furthermore, the global stiffness matrix is calculated using **ConstructGlobalStiffnessMatrix**. **ConstructSupportsAndLoadVectors** is used to construct a list of supports and a list of loads. This list of supports is then used to set the corresponding values in the stiffness matrix to zero to simplify the subsequent calculations. Since the stiffness matrix must be inversed, the diagonals of these DOFs are set to one, so as not to make the matrix singular. The next step in the process is to inverse the stiffness matrix and solve the equation $\mathbf{r} = \mathbf{K}^{-1}\mathbf{R}$. Lastly, the **CalculateResults** method is used to calculate the stresses and strains from displacement. The result is outputted with the custom result class, for post-processing.

4.1.3 Preview results component

These two components visualize the results from the **FEMSolver** component.

DisplacementPreview (NewNodes, displacements, scale):

The purpose of this component is to visualize the deformation of the beam in 3D. The component takes the displaced nodes, the displacements and a scaler and uses this to create a deformed mesh. Looping through all displaced points, the displacement is multiplied by the scaler and added. This magnifies the displacement and is especially useful for cases with small displacements. The new points are used to create a mesh which, in turn, is colored after the amount of displacement for each element. With a large displacement colored red and a small displacement colored green, see Figure 5.3 for an example.

StressPreview (NewNodes, Stress, stressDirection):

This component has the calculated stresses in each node from the FEM solver component as input, along with the displaced nodes and an integer slider to select type of stress to visualize. In

this component, the von Mises stresses are calculated using Equation (2.25). The output of this component is a colored mesh. To give each element a color, the average value is calculated from the internal nodes of the elements. This value is translated into a color. The colors are determined from a red-green color map, where the element with the highest value turns red, and the one with the lowest value turns green. By connecting an integer slider to the component, the user can decide which result values they want to visualize. The slider can be adjusted in the range from 0 to 6, where 0, 1 and 2 give the stress in the local xx, yy, and zz direction. 3, 4, and 5 give the shear stress in the local xy, yz, and zx directions, and the integer 6 will give the von Mises stress. The output of the component is a colored mesh and a list of stresses, which is chosen by the integer slider.

4.2 SolidFEM Plugin with 20-node Hex Element

This chapter includes a description of the Grasshopper plugin *SolidFEM* created by Marcin Luczkowski and Sverre Magnus Haakonsen. The authors of this thesis contributed with further development of the plugin, with the intention of implementing elements with higher-order shape functions. Therefore, this is a continuation of the first simple FEM Solver plugin described, which was used as an introduction to FEM with solid elements in AAD. The theory behind this implementation is described in Section 2.1.6. The benefits of using elements with more nodes come from the ability to represent quadratic shape functions. This is especially beneficial for representing bending modes and more complex geometry. Elements with higher-order shape functions are therefore preferred when analyzing connections and other details. Although the computational cost for one element increases substantially when these mid-side nodes are added, fewer elements are needed for convergence.

The SolidFEM plugin is fully developed for the 8-node hex element; the task was therefore to implement the 20-node hex element in the already established components. In addition to editing and expanding existing components, a new component was created. All code involving the 20-node elements have been created by the authors of this thesis. Some of the existing code has been tweaked to support the implementation of the 20-node element. Table 4.3 lists the components of the plugin and Figure 4.3 shows a flowchart of how the different components work together. Table 4.4 lists all custom classes in the plugin, with a short description.

This method chapter will describe all components of the plugin, but will mainly focus on the sections regarding implementation of the 20-node hexahedron element. The first two subsections describe three classes that contain all support methods for the other components. Thereafter, the main components of the plugin are described.

Components of the plugin
NEWFEMSolver
FEMLoadMesh
FEMBoundaryOnPointsMESH
FEMMaterial
MeshPreview
ConvertMeshTo20Node

Table 4.3: All components of the SolidFEM plugin.

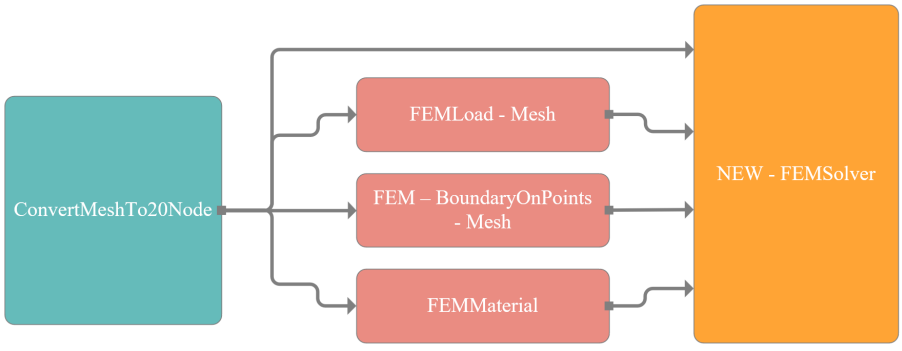


Figure 4.3: Flowchart of the SolidFEM plugin.

Class name	Properties	Description
Element	id, name, Nodes, Connectivity, Type, MeshQuality, ElementMesh, TypologyVertices, LocalK, LocalB	The element class is used to describe the FEM element, with its main features being the nodes, connectivity and type.
Node	ID, Name, Coordinate, BC-U, BC-V, BC-W, Type	The Node class is used to describe each individual node in the element, with its ID, Name, Coordinate, Boundary Condition in three directions, as well as the type (edge, corner or internal).
Material	E, PossionRatio, YieldingStress, Weight	The material class describes the material used in the FEA, with its features being the Young's modulus, the Poisson's ration, yielding stress and self-weight.
Support	Position, Tx, Ty, Tz	The support class describes the boundary conditions in three directions of a given point.
FEMUtility		This class contains most of the methods used in the plugin.
FEMMatrices		This class contains the methods for creating the different stiffness matrices used.
GrahamScan		This class contains methods for doing Graham Scans on the meshes.

Table 4.4: All classes for the SolidFEM plug-in.

4.2.1 FEMUtility and FEMMatrices

These two classes contain most of the methods used in the plugin, where most of these methods are found in the **NEWFEMSolver** component. Starting with the **FEMUtility** component, it contains twelve different methods that are described in the following.

GetMeshNodes (meshList): This method has a list of meshes as input. By looping through all the meshes in the list, the method creates a new list of unique nodes from the mesh. This is done by controlling the distance of each new point from the points already added to the list of nodes.

$$\textbf{return:} \begin{bmatrix} x_0 & y_0 & x_0 \\ x_1 & y_1 & x_1 \\ \vdots & \vdots & \vdots \\ x_{n_{pts}} & y_{n_{pts}} & x_{n_{pts}} \end{bmatrix}$$

ElementsFromMeshList (meshList, globalNodePts): This method has a list of meshes and a list of unique global node coordinates as input. The method uses these inputs to create elements. Looping through all the meshes in the list, a new element is made from each mesh. It gets an ID, connectivity, and a list of nodes. To get the correct global ID for the nodes (which also is the connectivity), the distance between each node in the element and the points in the input globalNodePts are checked. The element also gets assigned a type depending on the number of nodes. Either *Hex8* for eight nodes, *Hex20* for 20 nodes, or *Tet4* for four nodes.

$$\textbf{return:} \begin{bmatrix} \text{Element}_0 \\ \text{Element}_1 \\ \vdots \\ \text{Element}_{n_{els}} \end{bmatrix}$$

LocalCartesianCoordinates (element): The input of this method is an element. It moves all points in the element, so the first node has the coordinates $(0,0,0)$.

$$\textbf{return:} \begin{bmatrix} x_0 & y_0 & x_0 \\ x_1 & y_1 & x_1 \\ \vdots & \vdots & \vdots \\ x_{n_{nodes}} & y_{n_{nodes}} & x_{n_{nodes}} \end{bmatrix}$$

GetGaussPointMatrix (order, elType): This method takes an order as input in addition to the element type. The order represents the number of Gauss integration points in each direction, as illustrated in table 2.1 with the letter n. Based on the input, a matrix of Gaussian integration points is returned as output. The method uses an if check based on the element type to select the correct number of Gauss points. For most of the element we only have one set of Gauss points, the matrix returned is therefor straight forward. For the 20-node hexahedron element the method creates a matrix for two or three integration points in each direction, depending on the input order.

$$\textbf{return:} \begin{bmatrix} \xi_0 & \eta_0 & \zeta_0 \\ \xi_1 & \eta_1 & \zeta_1 \\ \vdots & \vdots & \vdots \\ \xi_n & \eta_n & \zeta_n \end{bmatrix}$$

PartialDerivateShapeFunctions (r, s, t, elType): By inputting a set of Gaussian coordinates, r, s and t, and the type of element, this method creates a matrix with the partial derivatives of the shape functions. For the 8-node hexahedron and four-node tetrahedron element, the approach is simple with a direct assignment of the matrix. For the 20 - node hexahedron element, a couple more steps are needed. An empty matrix with the correct shape is created. In addition, four lists of indexes are created that correspond to the type of nodes (corner or mid-side). Then a matrix of the natural coordinates is created. Looping through the rows of this matrix, the different values of the result matrix are calculated using the equations in Section 2.1.6.

return: $\begin{bmatrix} \mathbf{N}_\xi \\ \mathbf{N}_\eta \\ \mathbf{N}_\zeta \end{bmatrix}$

GetShapeFunctions (r, s, t, elType): This method uses the same approach as **PartialDerivateShapeFunctions**, to create a vector of shape functions. A set of Gauss coordinates, r, s, and t, and the element type are inputted. Then different methods are used for the different types of elements to create the shape functions. As for the derivated shape functions, the vector is created directly for the linear elements, while a more complex approach is used for the 20 - node element. This approach is the same as for the derivated shape functions, only with different equations; see Section 2.1.6.

return: $\begin{bmatrix} \mathbf{N} \end{bmatrix}$

DisplacementInterpolationMatrix (shapeFunctins, dofs): By inputting the shape functions vector, and degrees of freedom for each node, a matrix with the shape functions along the diagonal is created.

return: $\begin{bmatrix} \mathbf{N} & 0 & 0 \\ 0 & \mathbf{N} & 0 \\ 0 & 0 & \mathbf{N} \end{bmatrix}$

GetBodyForceVector (material, elements, numGlobalNodes): The inputs are the material, a list of the elements, and the number of unique global nodes. This method calculates a load vector for the self-weight. It begins by creating an empty load vector and the body load vector. Then it loops through all elements of the list. For each element, a matrix of global coordinates is created for each node. The order of Gauss integration is assigned, which is 3 for 20-node elements and 2 for the other supported elements. A matrix is created for the Gaussian coordinates with the **GetGaussPointMatrix** method. This matrix is looped through to perform numerical integration. For each set of Gauss coordinates, a matrix with the partial derivatives of the shape functions is created by the **PartialDerivateShapeFunctions** method. This matrix is multiplied by the coordinates of the global element to calculate the Jacobian determinant. As described by the theory in Section 2.1.3. Furthermore, a matrix for shape functions is created with the **GetShapeFunctions** method, as well as an interpolation matrix with the **DisplacementInterpolationMatrix** method. This matrix is multiplied by the body load vector to obtain the interpolated load vector in the Gauss points.

The only thing missing from numerical integration are the weights. These depend on the type of element used. Then, the load vector is calculated and added to the list of load vectors. Finally, the element load vector is added in the correct position in the global load vector. This global load vector is outputted.

return: $\begin{bmatrix} S^0_{F1} \\ S^0_{F2} \\ \vdots \\ S^0_{Fn} \end{bmatrix}$

CalculateDisplacementCSparse (K-gl, R-gl, applyBCToDOF): This method needs the global stiffness matrix, the global load vector, and a list of boundary conditions as input. Looping through the list of boundary conditions, the corresponding value in the stiffness matrix is set to zero for the whole row and column, except for the diagonal. The diagonal is set to 1, to avoid the stiffness matrix becoming singular and unable to be inverted. Then Equation 2.1 is solved and the displacements are outputted.

return: $\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}$

CalculateElementStrainStress (element, u, material): The **CalculatedElementStrainStress** method uses the displacement vector and material properties to calculate the stress and strain of a single element. First, the material matrix is created from a simple method in the material class. Then the B-matrix is calculated using the **CalculateElementMatrices** method, with reduced integration for the 20-node hexahedron element, since these points correspond to the optimal evaluation points. Furthermore, empty matrices for element Gauss strain and stress, in addition to element strain and stress, are made. Using the connectivity of each element, nodal displacements can be extracted from the global displacement vector. There are three approaches for further calculation, depending on the type of element *Hex8*, *Hex20*, or *Tet4*. The approaches for the hexahedral elements are similar. By looping through the list of B-matrices and multiplying each of these by the local deformation, the Gaussian strain and stress matrices are found. To obtain the nodal stress and strain, the strain and stress must be extrapolated from the Gauss points, as shown in Section 2.1.5. This is done by creating a matrix of extrapolation coordinates and looping through these. For each loop, the shape functions are found and multiplied by the Gauss strain and stress matrices, which yield the elemental nodal strain and stress. The difference between 8- and 20-node elements is that reduced integration is used for the 20 node. For the four-node tetrahedron element, a direct approach is used, since the B-matrix is constant.

return: $\begin{bmatrix} \varepsilon^0_x & \varepsilon^1_x & \dots & \varepsilon^{n_{nodes}}_x \\ \varepsilon^0_y & \varepsilon^1_y & \dots & \varepsilon^{n_{nodes}}_y \\ \varepsilon^0_z & \varepsilon^1_z & \dots & \varepsilon^{n_{nodes}}_z \\ \gamma^0_{xy} & \gamma^1_{xy} & \dots & \gamma^{n_{nodes}}_{xy} \\ \gamma^0_{yz} & \gamma^1_{yz} & \dots & \gamma^{n_{nodes}}_{yz} \\ \gamma^0_{zx} & \gamma^1_{zx} & \dots & \gamma^{n_{nodes}}_{zx} \end{bmatrix}, \begin{bmatrix} \sigma^0_x & \sigma^1_x & \dots & \sigma^{n_{nodes}}_x \\ \sigma^0_y & \sigma^1_y & \dots & \sigma^{n_{nodes}}_y \\ \sigma^0_z & \sigma^1_z & \dots & \sigma^{n_{nodes}}_z \\ \tau xy^0 & \tau xy^1 & \dots & \tau xy^{n_{nodes}} \\ \tau yz^0 & \tau yz^1 & \dots & \tau yz^{n_{nodes}} \\ \tau zx^0 & \tau zx^1 & \dots & \tau zx^{n_{nodes}} \end{bmatrix},$

CalculateGlobalStress (elements, u, material): This method uses **CalculateElementStrainStress** to calculate the global stresses. The inputs are a list of elements, the global displacement vector, and the material. When looping through all elements and calculating the strain and stress matrices with the **CalculateElementStrainStress** method, a global stress matrix is created. The connectivity of each element is used to add the stress of the element to the correct position in the global stress matrix. Since many nodes receive contributions from multiple elements, a nodal average is calculated. Furthermore, the von Mises stress in every node is calculated using Eq. 2.25. The same is done for an element average, i.e. the average stress over a whole element. The global stress matrix is outputted, together with the nodal Mises and the element Mises.

$$\textbf{return:} \begin{bmatrix} \sigma_x^0 & \sigma_x^1 & \dots & \sigma_x^{n_{nodes}} \\ \sigma_y^0 & \sigma_y^1 & \dots & \sigma_y^{n_{nodes}} \\ \sigma_z^0 & \sigma_z^1 & \dots & \sigma_z^{n_{nodes}} \\ \tau xy^0 & \tau xy^1 & \dots & \tau xy^{n_{nodes}} \\ \tau yz^0 & \tau yz^1 & \dots & \tau yz^{n_{nodes}} \\ \tau zx^0 & \tau zx^1 & \dots & \tau zx^{n_{nodes}} \end{bmatrix}, \begin{bmatrix} \sigma_m^0 \\ \sigma_m^1 \\ \vdots \\ \sigma_m^{n_{nodes}} \end{bmatrix}, \begin{bmatrix} \sigma_m^0 \\ \sigma_m^1 \\ \vdots \\ \sigma_m^{n_{elems}} \end{bmatrix}$$

The FEM-Matrices class contains the two methods described in the following.

CalculateElementMatrices (element, material, intType): This method calculates the local stiffness matrix and the **B**-matrix. The inputs are an element, a material, and the integration type, full or reduced. First, the material matrix is found. Then empty stiffness- and B-matrices are made. Looping through the nodes of the element, a matrix of global coordinates is created. There are different methods for the three types of supported elements. This section addresses the method for the 20-node elements, but the other methods are similar. The method is explained in detail in Section 2.1.3. For the 20-node method, both full and reduced integration can be used; the only difference is the number of integration points, 2x2x2 or 3x3x3. The first thing that is needed is a matrix with Gaussian integration points, which is found using the **GetGaussPointMatrix** method. These points are looped through. For each set of coordinates, a matrix with the partial derivatives of the shape functions is found using the **PartialDerivateShapeFunctions** method. This matrix is multiplied by the global coordinates to obtain the Jacobian matrix. Furthermore, the derivatives of the shape functions in Cartesian coordinates are found by multiplying the inverse of the Jacobian matrix by the matrix with partial derivatives. In addition, the determinant of the Jacobian matrix is found. The **B**-matrix is established with a direct approach. Lastly, the stiffness matrix is calculated using Eq. 2.18, with weights found in Table 2.1. The local stiffness matrix and the **B**-matrix are outputted.

$$\textbf{return:} \begin{bmatrix} N_{,x} & 0 & 0 \\ 0 & N_{,y} & 0 \\ 0 & 0 & N_{,z} \\ N_{,y} & N_{,x} & 0 \\ 0 & N_{,z} & N_{,y} \\ N_{,z} & 0 & N_{,x} \end{bmatrix}, \begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{nn} \end{bmatrix}$$

GlobalStiffnessCSparse (elements, numNode, material): With a list of elements, the number of nodes, and the material inputted, this method calculates the global stiffness matrix. An empty stiffness matrix is created. For each element in the list of elements, the local stiffness matrix is calculated using the **CalculateElementMatrices** method. The local stiffness contribution is then

added to the global stiffness matrix at the correct index using the connectivity of each element. In addition, the **B**-matrix is calculated and added as a property to the element. The global stiffness matrix is outputted.

return: $\begin{bmatrix} K_{11} & K_{12} & \cdots & K_{1n} \\ K_{21} & K_{22} & \cdots & K_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ K_{n1} & K_{n2} & \cdots & K_{nn} \end{bmatrix}$

4.2.2 GrahamScan

This class contains a couple of different methods. The only one of these that is called outside the class is **DoGrahamScan**, which utilizes some of the other methods. The purpose of a Graham scan is to sort the nodes in an element in the correct order. This order is important for the following analysis to work. The method takes a mesh as input and checks the position for all nodes and faces. Then it orders them in the correct way, as shown in Figure 2.1.

4.2.3 ConvertMeshTo20Node component

This component has a list of meshes as input and a list of meshes as output. The purpose of the component is to add the mid-side nodes for the 20-node hexahedron element. As described in section 2.1.6, the 20-node hexahedron element is equivalent to the 8-node hexahedron element, only with nodes added in the middle of all edges. To achieve this, the component first loops through all meshes. For each mesh, a Graham scan is performed, using the **DoGrahamScan** method; this ensures that the 8-node mesh is correctly ordered. Then the mid-side nodes are added in the correct order, as shown in Figure 2.2, taking the average of the correct corner nodes to get the points between them. This creates a mesh with 20 nodes.

4.2.4 FEMBoundaryOnPointsMESH

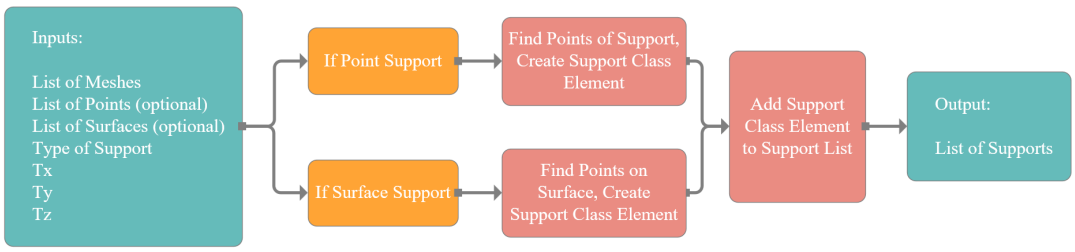


Figure 4.4: Flowchart of the FEMBoundaryOnPointsMESH component.

This component takes a list of meshes, a list of points, a list of surfaces, the type of support, and three Boolean values for the three directions of translation. The component supports two kinds of boundary condition, either for a list of explicit points on the mesh or for all points on a surface. First, the mesh is cleaned and sorted using a **GrahamScan**, but only if it is an 8-node mesh, since

the 20-node mesh has been cleaned and sorted in the **ConvertMeshTo20Node** component. Then the unique nodes of the mesh are found using the **GetMeshNodes** method. There are different methods for point and surface support. To create the support list for the point support, a loop is created through the points in the point list. For each point, a node index is found by using the **GetClosestNodeIndex** method. This method compares a point with the list of unique nodal points to obtain the correct global index. Then, a support class element is created and added to the support list. For surface support, a loop is created through all surfaces in the surface list. For each surface, all the points in the list of unique nodal points are checked to find the points on the surface. Then, a support class element is created for all these points and added to the support list. This component outputs a list of support class elements.

4.2.5 FEMLoadMesh

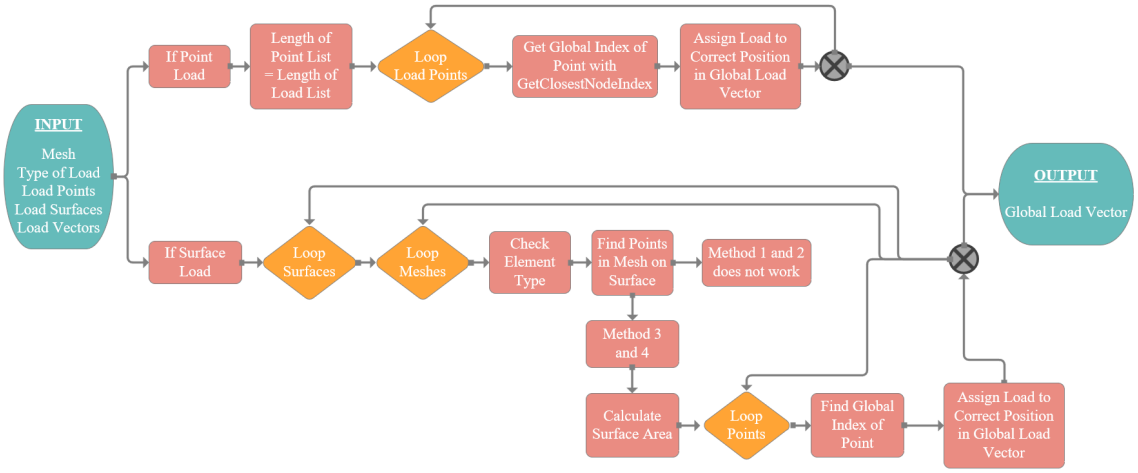


Figure 4.5: Flowchart of the FEMLoadMesh component.

The load component has a list of meshes, the type of load, a list of points for point loads, a list of surfaces for surface load, and a list of load vectors. As for the support component, the first step is to clean and order the meshes, but only for the 8-node, since the 20-node already has been cleaned. Then, a list of unique nodal points is created with the **GetMeshNodes** method, as well as an empty vector for the residual forces. The approach to creating the load vector depends on the type of load.

For point loads, the approach is straightforward. First, there is a check to control that the list of points and the list of load vector has the same number of elements; if not, the last element in the load vector list is duplicated so that the lengths of the two lists are the same. The next step is to loop through all points. The **GetClosestNodeIndex** method is used to find the correct index for the loading points. The load vector is then deconstructed and added to the correct position in the residual force list, using the global index of the loaded point.

For surface loads, the approach is more complicated. First, it varies according to the type of element used. This section will describe the method used for the 20-node element. All methods loop through all loaded surfaces and then through all meshes in the mesh list. For the 20-node element, there are two different methods of load lumping. Regardless of the method used, the first

step is to create a list of points in the mesh that are on the loaded surface. The two methods use the same approach, with load lumping. The only difference is that Method 2 uses $-P/12$ as a load for the corner nodes and $P/3$ for the mid-side nodes, while Method 2 uses $P/8$ for all nodes. To calculate the global load vector, the area of the loaded surface is calculated. Then, nodal load vectors are created by looping through the loaded points and assigning the correct load values. Finally, the global node index is found using the **GetClosestNodeIndex** method, and the loads are assigned to the correct position in the global load vector.

4.2.6 FEMMaterial

This component has the Young's modulus, the Poisson's ratio, the yielding stress, and the weight of the material as input. The simple component creates a material class element with the correct properties and returns it.

4.2.7 NEWFEMSolver

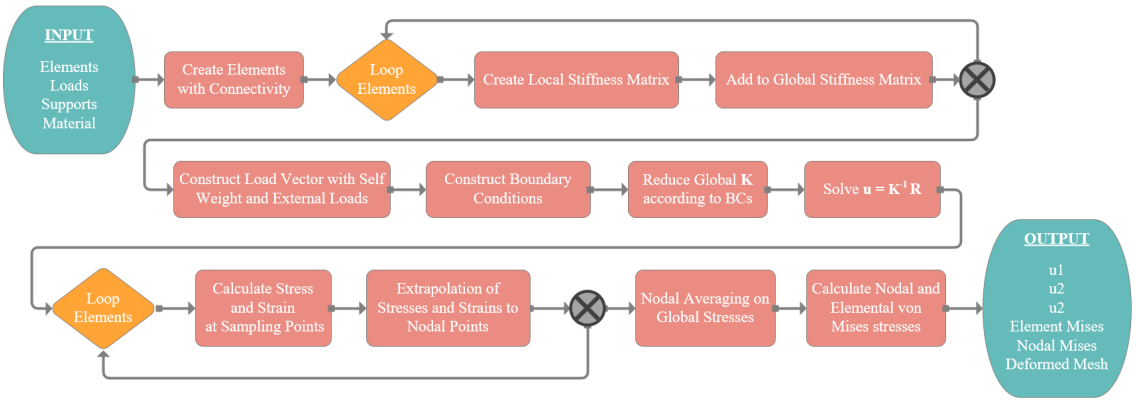


Figure 4.6: Flowchart of the FEM Solver component.

The FEMSolver is the main component of the plugin. It takes as input a list of meshes, a list of loads, a list of supports, and a material. The output is three lists of displacements in the global x, y, and z directions, in addition to a list of element Mises stresses, nodal Mises stresses, and the calculated mesh. As for the other components, the first step is to clean and sort the mesh if it is 8-noded. Then a list of unique nodal points is created using the **GetMeshNodes** method, as well as a list of unique nodes. After this, a list of elements is created using the **ElementsFromMeshList** method. The **GlobalStiffnessCSparse** method is used to create the global stiffness matrix. To solve the force-displacement relation in Eq. 2.1, the global load vector is needed. This consists of the self-weight and external loads. The self-weight is calculated with the **GetBodyForceVector** method, while the external loads are an input. These two load vectors are summed to create a total load vector. Then a list of boundary conditions is created. The **FixBoundaryConditions** method uses the list of supports and the list of unique nodal points to create a list of Boolean values that represents a fixed or free point for each unique nodal point in the mesh. The method converts

the support class elements into a list of Booleans. The stiffness matrix, the load vector and a list of boundary conditions are used in **CalculateDisplacementCSparse** to calculate the global displacement vector. This vector is used in **CalculateGlobalStress** to calculate global stresses, in addition to the nodal and element von Mises stress. The last step is to prepare the different outputs of the lists to achieve a clear result in Grasshopper.

4.2.8 MeshPreview

This component has the resulting mesh from the solver as input, in addition to a type and scaler. The type determines what the mesh should be colored after; displacement, Mises stress, utilization, or stresses in the xx, yy, or zz directions. First, the displacements are scaled according to the scaler input. This is done by multiplying the displacement in the three directions with the scaler value and then creating new coordinates for the vertices in the mesh. Then, depending on the type of result desired, the mesh is colored accordingly.

4.3 Machine Learning

In this Section we describe the general workflow of our ML models. An introduction to the different classes and functions will be provided with a simplified and generalized code example. A more detailed explanation of each case will be provided in the Case Study chapters. In this section, we focus on presenting the ML workflow with main focus on the five files: **dataset.py**, **NN.py**, **trainer.py**, **run.py** and **sweep_run.py**.

4.3.1 Creation of datasets

To train an ML model, a large collection of data with input features and targets is needed. The creation of these datasets differs between the different study cases. Some of the datasets were created in Visual Studio Code with Python, while others were created in Grasshopper. An in-depth explanation of the different approaches to data creation can be found in the relevant case study sections.

4.3.2 Weights and Biases

Weights and Biases (WandB) has been used as a platform to log all projects and runs. The platform is a helpful tool in collaboration, to get an overview of the projects and to recreate previous network configurations. With WandB, it is possible to log all information needed to recreate a run, for example, all hyperparameters and network depths and widths. In addition to information about the run, all interesting results are logged. In most cases the results logged are the training loss and the validation loss, in some cases we also log an image representation of the target and prediction. Figure 4.7 illustrates the result overview page at Wandb with some selected graphs and images.

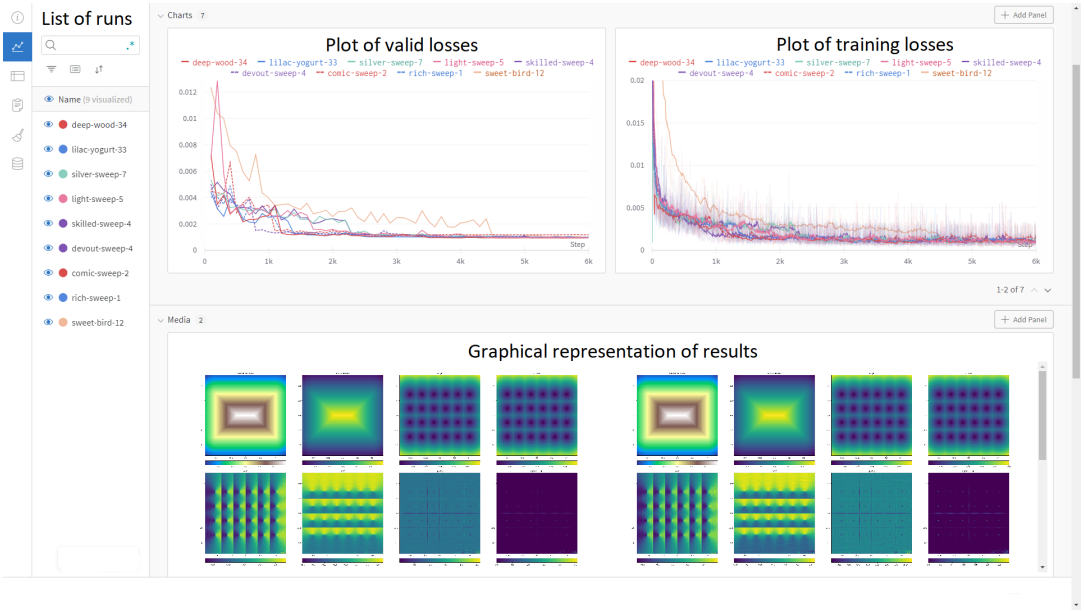


Figure 4.7: Snip of an example project in WandB.

4.3.3 Dataset

This file contains the **CustomDataset** class that is used to load the data in the run file. The class contains three functions, where the first one is called **init** and initializes the data directory. The second function, **len**, returns the number of samples in the dataset. The last function, **getitem**, loads and returns a sample from the dataset at a given index, idx. Based on this index, the location on the disk is identified.

```
class CustomDataset(Dataset):

    def __init__(self, root_dir="data", split="train"):
        self.data_path = f"{root_dir}/{split}"
        self.data = [folder for folder in os.listdir(self.data_path)]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        folder_name = self.data[idx]
        full_path = f"{self.data_path}/{folder_name}"
        data = torch.load(f"{full_path}/data.pth")
        x = data["x"]
        y = data["y"]
        return x, y
```

Listing 4.1: Simplified code of dataset.py.

All data are collected in a folder named data, and the location of this folder is set as the root directory in the **init** function. The data folder contains three subfolders named train, validation, and test, all of which contain different data samples. By default, the **init** function uses train at the split position, to load data from the test or validation folder, simply change the split argument to either of these strings. In this simple illustration code, shown in Listing 4.1, it is assumed that each sample is a .pth file that contains a dictionary with the input x and the target y, which is already in Pytorch format. This is not always the case; therefore, some manipulation and sorting might be necessary after loading the data.

4.3.4 NN

The main class in this file is the **NeuralNet** class itself. In some of the case studies, this file also contains some helper classes, but these classes will be explained in depth when we come to the relevant case studies. For now, we focus on the **NeuralNet** class. To illustrate this class, a simple fully connected network is used, similar to the network in Figure 2.5. This example is shown in Listing 4.2

```
class NeuralNet(nn.Module):
    def __init__(self, layer_sizes, device="cuda"):
        super(NeuralNet, self).__init__()

        self.device = device
        layers = []
        input_features = 4

        for output_features in layer_sizes:
            layers.append(nn.Linear(input_features, output_features))
            input_features = output_features
        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        pred = self.layers(x)
        return pred
```

Listing 4.2: Simplified code of NN.py.

The example in Figure 2.5 has four input features, three layers with three neurons in each layer, and two output features. The layer sizes, in this case [3 3 3 2], is a list that is sent into the class from the **run** file. The last item in the list corresponds to the output layer, which is fixed. To change the depth of the network, we simply add or remove items in the list, and to change the width or number of neurons in each layer, the value of each item in the list is changed. In this case, the default device is set to "cuda", but this property is also sent into the class from the run file, where an availability check is performed. We define the network by subclassing nn.Module from Pytorch and initializing the layers in the **init** method. In the forward pass method, every nn.Module subclass implements the operations on the input data, before the prediction is returned.

4.3.5 Trainer

Trainer is the file that is used to train our model. The **train** function is the main function in this file; this function is called from the **run** file and takes a model, batch size, the three data loaders, the learning rate, and the device as input. An example code is shown in Listing 4.3

```
def no_grad_loop(data_loader, model, batch_size, device="cuda"):
    loss = 0
    cnt = 0
    for x, y in data_loader:

        # transfer data to device
        x = x.to(device)
        y = y.to(device)

        # forward pass
        pred = model(x)
```

```
            loss += F.l1_loss(pred, y)
            cnt += 1

    return loss/cnt

def train(model: NeuralNet, num_epochs, batch_size, train_loader,
            test_loader, validation_loder, learning_rate, device="cuda"):
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    # training loop
    iter = 0
    for epoch in range(num_epochs):
        loader = tqdm.tqdm(train_loader)
        for x, y in loader:

            #transfer data to device
            x = x.to(device)
            y = y.to(device)

            # forward pass
            pred = model(x)
            loss = F.l1_loss(pred, y)

            # backward and optimize
            optimizer.zero_grad()   # clear gradients
            loss.backward()         # calculate gradients
            optimizer.step()        # update weights
            iter += 1

            if (iter+1) % 50 == 0:
                # validation loop
                model = model.eval()
                valid_loss = no_grad_loop(validation_loader,
                                          model, batch_size, device="cuda")
                model = model.train()

    # test loop
    test_loss = no_grad_loop(test_loader, model, batch_size, device="cuda")
```

Listing 4.3: Simplified code of trainer.py.

The **train** function loops through a selected number of epochs and loads a batch of training data from the train loader. In each loop, the data are transferred to the correct device, the input features are sent into the model, and a prediction is returned. The prediction, together with the label, is used to calculate the loss. After this, the gradients are cleared from the last run, new gradients are calculated, and the weights are updated. This is the training process that will slowly improve a healthy model. As can be seen in the code, in Listing 4.3, a validation loop is performed for every 50th iteration. The validation loop uses the **no_grad_loop**. This does not calculate the gradients, nor does it update the weights. As described in the theory chapter, the validation loop does not

train the model, but is used to evaluate performance during training. The validation loop returns the average loss over the entire validation set. After all epochs are completed, a test loop is applied, using the same **no_grad_loop**, to get the performance of the trained network. For simplicity the code that logs data to WandB is excluded from this code, but it is worth mentioning that all losses, targets and predictions are logged as graphs and in some cases image illustrations. An example of logged images can be found in Table B.1, where all logged images from a single run is used to create a GIF. In some of the case studies, early stopping and saved checkpoints are also included. Early stopping is implemented so that the trained model is returned if the model has converged. A check point code is included in the training loop; this code overwrites the saved check point if the model has improved with respect to the validation loss. As a result, the best performing model is saved as a check point. Additionally, a scheduler is used to reduce the learning rate when the validation loss begins to converge. The scheduler used is called *ReduceLROnPlateau*. A patience, denoted n here, is defined. Every time the validation loss does not decrease n consecutive times, the learning rate is multiplied with 0.1.

4.3.6 Run

The **run** file is used to start training the model. In the simplified code, illustrated in Listing 4.4, the example from Figure 2.5 is used. Before the **run** function, we also have some code, not included here, that sets the random seed. The weights in the network is initialized with random values; by setting the seed to a fixed number these random values are always the same. This enables us to reproduce and compare runs in a better way because we always have the same starting point. Also not included is the code that saves the trained model and gives the opportunity to start a new run from the saved model.

```
def run():
    # Hyperparameters
    layer_sizes = [3,3,3,2]
    num_epochs = 50
    batch_size = 16
    learning_rate = 0.001

    # dataset
    train_dataset = CustomDataset()
    test_dataset = CustomDataset(split="test")
    validation_dataset = CustomDataset(split="validation")

    # dataloader
    train_loader = DataLoader(train_dataset, batch_size)
    test_loader = DataLoader(test_dataset, batch_size)
    validation_loader = DataLoader(validation_dataset, batch_size)

    model = NeuralNet(layer_sizes).to(device)
    train(model, num_epochs, batch_size, train_loader, test_loader,
                                validation_loader, learing_rate, device)
```

Listing 4.4: Simplified code of run.py.

After setting the hyperparameters, **CustomDataset** retrieves the features and labels of our dataset one sample at a time. By using Dataloader, we can pass samples of minibatches into the model and reshuffle the data at every epoch during training, which reduces the possibility of overfitting. We also use multiprocessing to speed up data retrieval. The model is imported and transferred to the device. Finally, we call the **train** function and send in the required input.

4.3.7 Sweep_run

Sweep_run is essentially the same component as **run**, with the only difference being that a handful of hyperparameters that we want to optimize are extracted from a configuration file created in Wandb. This configuration file is a file with many different hyperparameters that we want to test on our network. For example, the file can contain many networks with different depths and widths, a range of learning rates, dropouts, and the number of epochs. Basically, any variable we would like to optimize. The sweep then randomly selects and tests the configurations of these hyperparameters to find the configuration that best performs the task, often to minimize the validation loss. This process is called *Hyperparameter Tuning*, and Figure 4.8 shows the result of a tuning process.

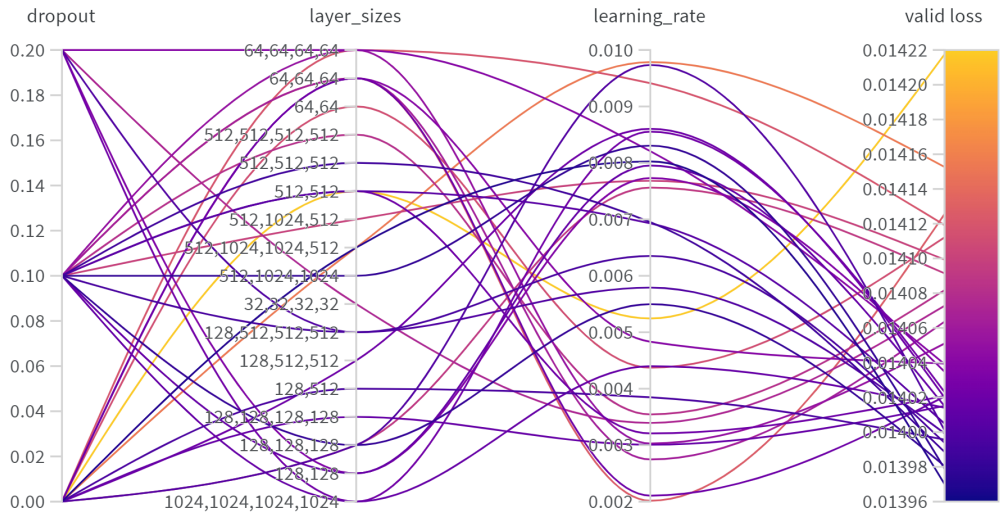


Figure 4.8: Example of sweep run in WandB.

5 Case Studies

5.1 Case study 1: Verification of the Simple FEM Solver plugin

To verify the accuracy of the solver, a simple cantilever beam, as illustrated in in Figure 5.1, is analyzed. The analysis is done in Grasshopper and the architecture of this analysis is described below. Furthermore, the results of the Simple FEM Solver are compared with the results of Abaqus. The dimensions of the cantilever and the total magnitude of the force are shown in Table 5.1, together with the material properties. A video showcasing this Case Study is found in Table B.1.

Width	Height	Length	∑ Loads	Young's Modulus	Poisson's ratio
200 mm	150 mm	1400 mm	-150 kN	210000 MPa	0.3

Table 5.1: Dimensions and material properties of the cantilever beam.

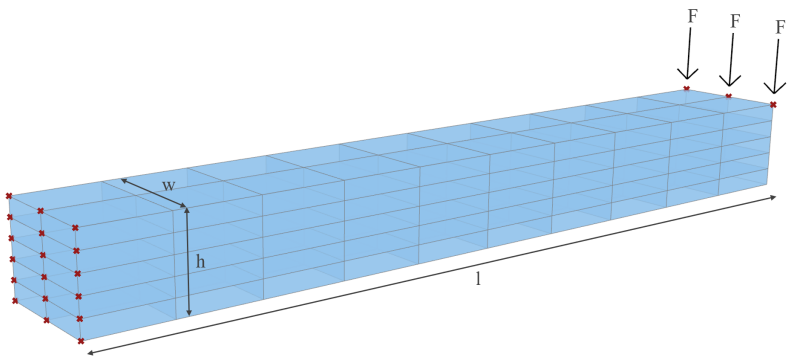


Figure 5.1: Model of the simple cantilever beam in Case Study 1 with 10x2x5 mesh.

5.1.1 Grasshopper

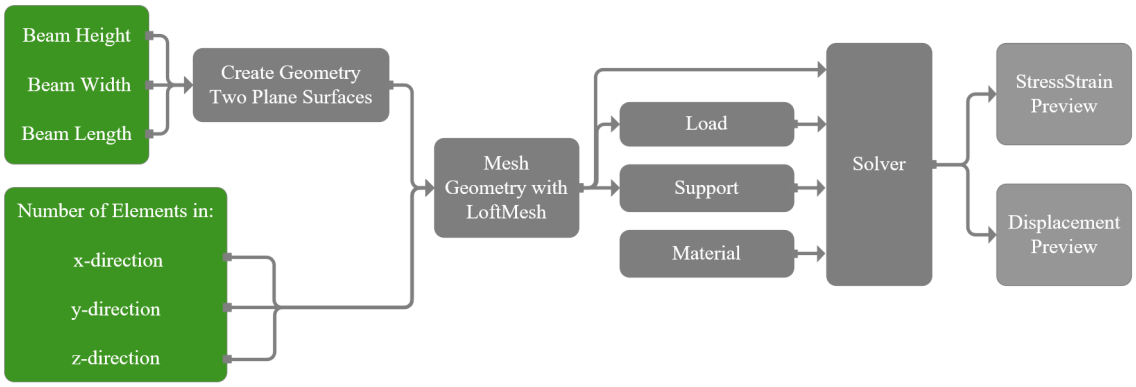


Figure 5.2: Flowchart of the cantilever beam in Grasshopper.

In Figure 5.2, a flowchart of the algorithm in Grasshopper is presented. To construct the cantilever as a parametric model in Grasshopper, the surface on the left end of the beam is first created. This surface sets the width and height of the cantilever. The input variables for the beam geometry are height, width, and length. These parameters are connected to number sliders so that they can

be easily adjusted. The surface is then duplicated and moved along the longitudinal direction (x-axis) by a vector where the x-value is the given length of the beam. The **Loft Mesh** component takes these two surfaces as input, along with integers which are the number of divisions in each direction, to create the beam as a uniform volumetric mesh. The number of divisions in each direction is equivalent to the number of elements in each direction. The **Loft Mesh** component creates 8-node elements.

The support points and load points are extracted from the mesh geometry. All nodes on the left end surface of the beam are set as support points, and the top nodes on the right end surface are used as load points. The total magnitude of the given force is distributed by the number of load points, and the corresponding load vectors are unit vectors in the negative z direction with this value. The **Load** component takes the load points and the load vector as input, and the **Support** component takes the support points and three Boolean toggles, set to true, to fix the nodes in all directions. Together with the material, these components are connected to the **Solver**, which performs the FEA. The results of the analysis are visualized with the **StressStrainPreview** and **DisplacementPreview** components.

Abaqus

To evaluate the results of our solver, the cantilever beam was reconstructed in Abaqus. The cross-section of the beam has been drawn in the sketch module and extruded with an extrusion depth equal to the length of beam. To mesh the part, three edges have been seeded to achieve the equivalent number of elements as the model in Grasshopper. In Abaqus, the same 8-node hexahedron element has been used, the C3D8 element. Steel was defined as a material and assigned to the part. The entire left end surface of the beam was set as a boundary condition, and point loads were applied to the top nodes on the right end surface.

5.1.2 Results

The results of the analysis are shown in the following section. Stress and displacement values are extracted from the yellow points in the figures. Figures 5.3 and 5.5 show the deformed model with a coloration that represents displacements and stresses. Tables 5.2 and 5.3 show the calculated displacements and von Mises stresses for both the Simple FEM solver and Abaqus, while Figures 5.4 and 5.6 show the displacement and stresses plots with respect to the number of elements in the mesh.

Displacements

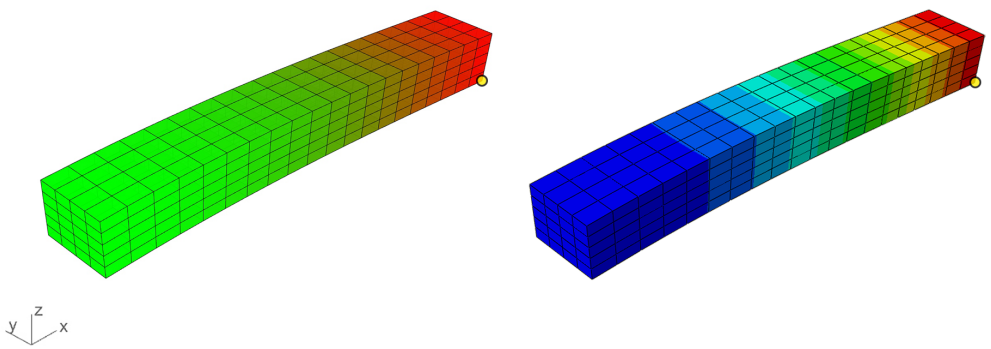


Figure 5.3: Study case 1, cantilever beam, displacement: Simple FEM Solver, 8-node hex (left) and Abaqus, C3D8 (right).

Mesh [x,y,z]	No. Elements	Simple FEM Solver (8-node)	Abaqus (C3D8)	Difference
10x1x1	10	-4.76 mm	-5.11 mm	6.85 %
10x2x5	100	-8.49 mm	-8.70 mm	2.41 %
20x4x5	400	-10.57 mm	-10.80 mm	2.13 %

Table 5.2: Displacement: Simple FEM Solver and Abaqus.

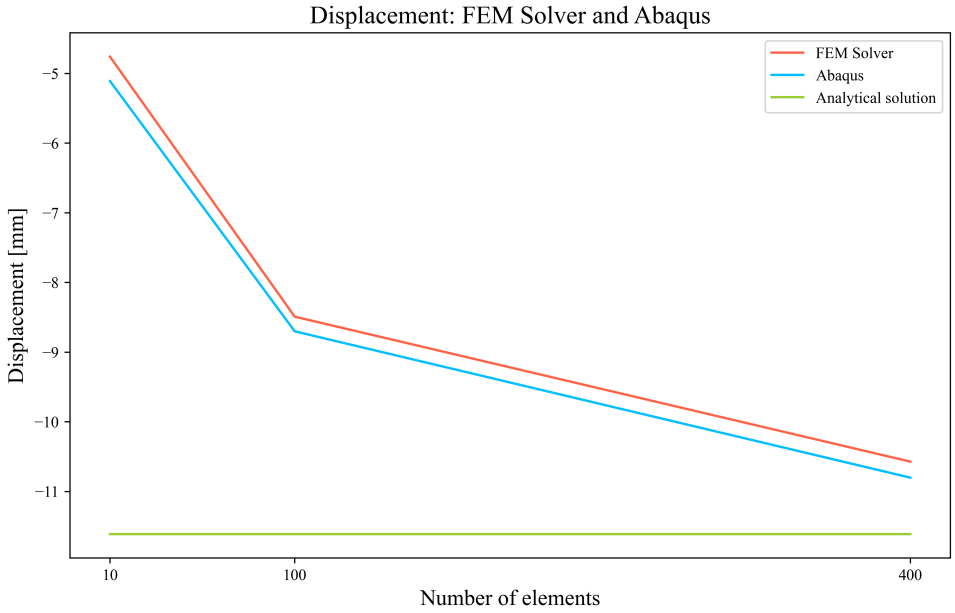


Figure 5.4: Displacement: FEM Solver, Abaqus and analytical.

Stress

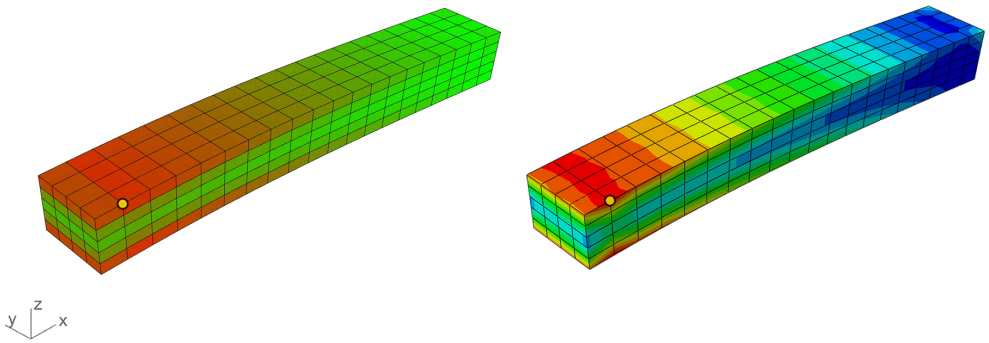


Figure 5.5: Study case 1, cantilever beam, von Mises stress: Simple FEM Solver, 8-node hex (left) and Abaqus, C3D8 (right).

Mesh [x,y,z]	No. Elements	Simple FEM Solver (8-node)	Abaqus (C3D8)	Difference
10x1x1	10	155 MPa	169 MPa	8.28 %
10x2x5	100	231 MPa	229 MPa	0.87 %
20x4x5	400	258 MPa	261 MPa	1.15 %

Table 5.3: Von Mises stress: Simple FEM Solver and Abaqus.

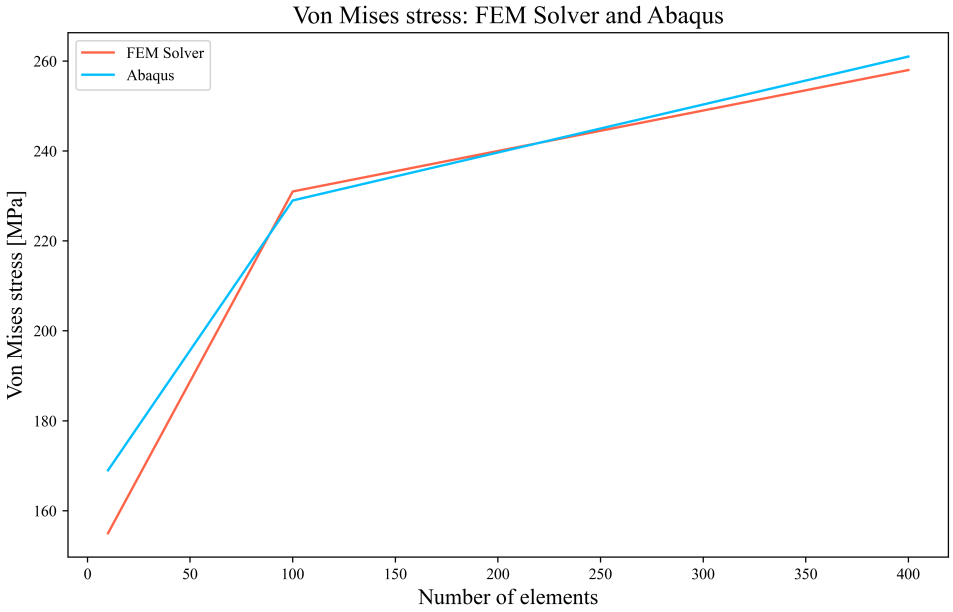


Figure 5.6: Stress: Simple FEM Solver and Abaqus.

5.1.3 Discussion

As Figure 5.4 shows, the displacements of both the Simple FEM solver and Abaqus are converging towards the analytical value of 11.61 mm, when the number of elements increases. A cantilever beam, such as this, is not the ideal case for the 8-node linear element because it is too stiff in bending due to shear locking. This case was chosen for simplicity, and since the same element is used in Abaqus, the comparison between the two is the important part. From the comparison with the analytical value in Figure 5.4, it is obvious that the 8-node element is too stiff. However, compared to the displacements in Abaqus, the results are acceptable.

Furthermore, the stress results are good compared to the Abaqus results. There are some differences between the results in Abaqus and the FEM solver, but these are small and would probably come down to more post-processing of the results in Abaqus. This could also be the reason why the difference between the FEM Solver and Abaqus reduces when the number of elements is increased. This case was used as an introduction to develop an FEM solver in Grasshopper. For this purpose, it performs well.

5.2 Case study 2: Verification of the SolidFEM plugin with 20-node elements

To verify the precision of the SolidFEM plugin, two different models were analyzed in Grasshopper. The first is a simple cantilever beam, while the second is an arched beam. The results have been compared with results from the SolidFEM plugin with an 8-node element, Abaqus, and for the cantilever beam with the analytical solution. The analysis in Abaqus has been done with the same 20-node hexahedron element as in the SolidFEM plugin. A video showcasing the arch beam in this Case Study is shown in Table B.1.

5.2.1 Cantilever beam

The first model analyzed is a simple cantilever beam; dimensions and material properties are shown in Table 5.4. The beam has been loaded with point loads at the right end and is fixed at the entire left end.

Width	Height	Length	∑ Loads	Young's Modulus	Poisson's ratio
100 mm	280 mm	3000 mm	-100 kN	210000 MPa	0.3

Table 5.4: Model: Dimensions and material properties of the cantilever beam.

Grasshopper

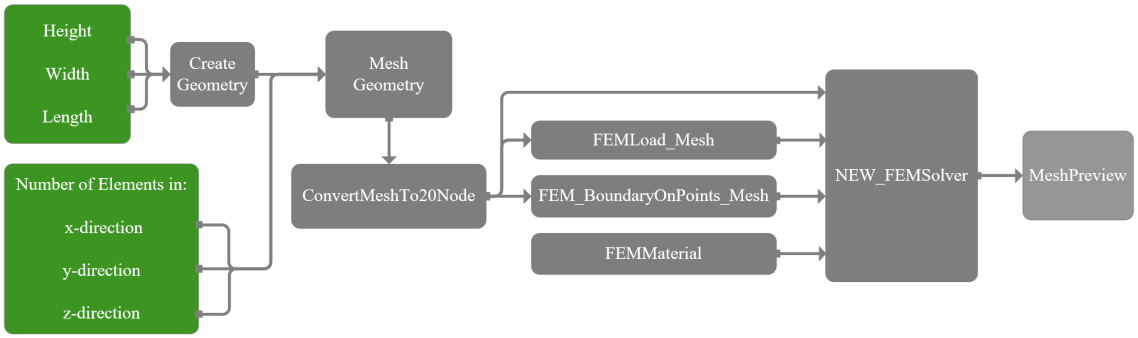


Figure 5.7: Flowchart of the Cantilever beam Grasshopper file.

A flow chart of the Grasshopper file is shown in Figure 5.7. For the parametric model in Grasshopper, the structure is quite simple. The input parameters for the length, height and width of the beam are connected to sliders, as well as the number of elements in the x, y and z directions for meshing. These values are then used to create boxes that each represent one element in the mesh and together create the whole geometry. These boxes are meshed to create a list of 8-node meshes. The **ConvertMeshTo20Node** component is used to create the 20-node mesh. Points and surfaces for loads and supports are created from the initial geometry. These are inputted, together with the mesh list, to create a load vector in the **FEMLoadMesh** component and the boundary conditions in **FEMBoundaryOnPointsMESH**. Together with the material, the load vector, the boundary conditions and the mesh list are inputted into **NEWFEMSolver** to perform the FEA. The results are previewed with the **MeshPreview** component. To compare the results in a consistent way, the displacements and stresses for the same node were sorted from the results.

Abaqus

For the best comparison between the analysis in Abaqus and Grasshopper, the Abaqus model was made to mimic the Grasshopper model. To create the geometry, an extrusion method was used. Then, the same mesh as in Grasshopper was applied for each step of refinement. A 20-node hexahedron element was used, which is called C3D20 in Abaqus. To mesh the model in the same way as in Grasshopper, the approach was to seed three edges, in the vertical, transversal, and longitudinal direction, with the correct amount of elements. The model was loaded with three-point loads at the right edge and fixed at the entire left end. Since Abaqus is a well-established and developed software, it was a point to make the analysis as simple as possible, since the SolidFEM plugin is a more raw software.

Results

As for Case Study 1, stress and displacement have been extracted from the same node in Grasshopper and Abaqus. This node is shown with a yellow point in the figures. Figures 5.8 and 5.10 show the resulting geometry with colored displacement and stresses. The numerical results of the analysis are shown below, with Table 5.5 showing the displacements and Table 5.6 showing the stresses. Figures 5.9 and 5.11 shows the plots of displacement and stresses with respect to the number of elements. The analytical solution for the displacements was calculated with the following equation:

$$w = \frac{PL^3}{3EI} \tag{5.1}$$

Where w is the vertical displacement, P is the point load, L the beam length, E the Young's modulus, and I the moment of inertia.

Displacement

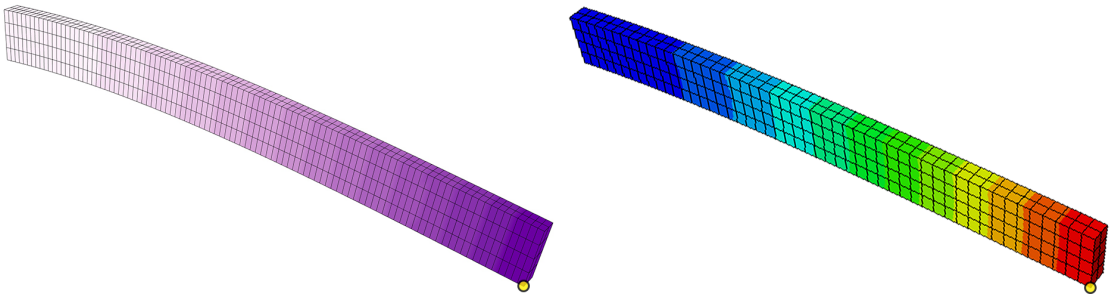


Figure 5.8: Study case 2, cantilever beam model with displacements. Grasshopper (Hex 20) to the left and Abaqus (C3D20) to the right.

Mesh	No. Elements	SolidFEM 20-node	SolidFEM 8-node	Abaqus (C3D20)
1x2x4	8	-17.92 mm	-0.54 mm	-17.53 mm
2x2x4	16	-22.19 mm	-1.99 mm	-21.68 mm
5x2x4	40	-23.59 mm	-8.54 mm	-23.30 mm
10x2x4	80	-23.88 mm	-16.36 mm	-23.33 mm
25x2x4	200	-24.01 mm	-22.15 mm	-23.48 mm
50x2x4	400	-24.05 mm	-23.38 mm	-23.52 mm
100x2x4	800	-24.06 mm	-23.72 mm	-23.54 mm

Table 5.5: Displacement: SolidFEM and Abaqus.

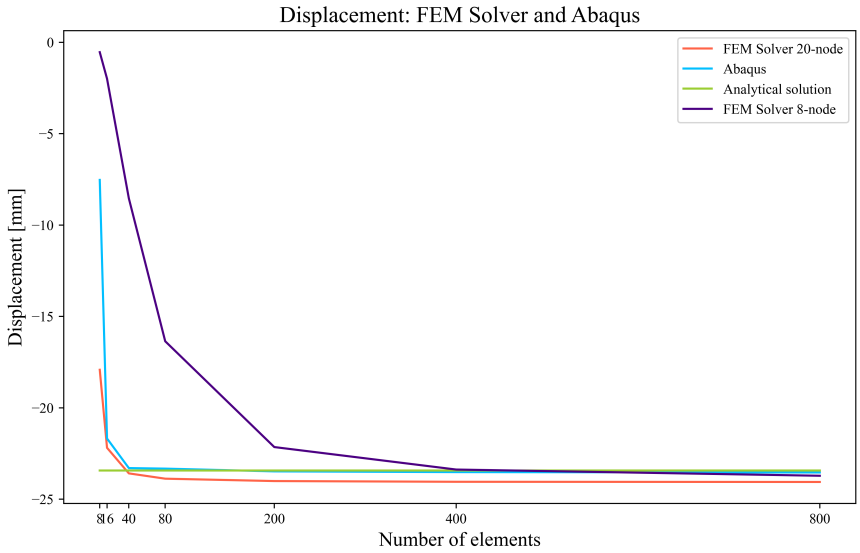


Figure 5.9: Study case 2, cantilever beam, displacement: SolidFEM, Abaqus and analytical solution.

Stress

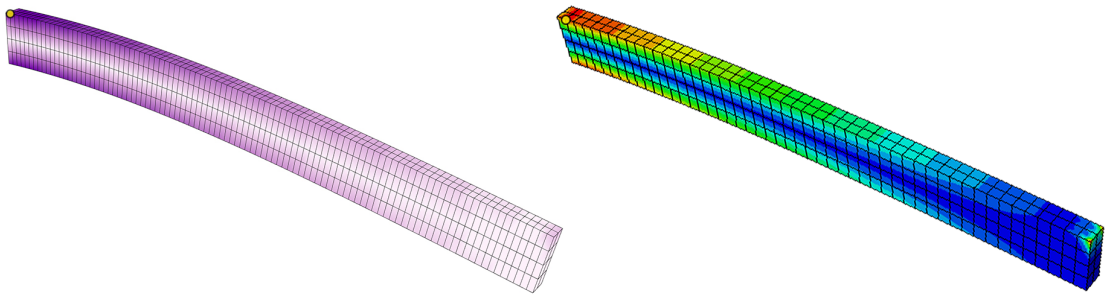


Figure 5.10: Study case 2, cantilever beam model with stress. Grasshopper (Hex 20) to the left and Abaqus (C3D20) to the right.

Mesh	No. Elements	SolidFEM 20-node	SolidFEM 8-node	Abaqus (C3D20)
1x2x4	8	110 MPa	19 MPa	114 MPa
2x2x4	16	163 MPa	59 MPa	170 MPa
5x2x4	40	200 MPa	126 MPa	204 MPa
10x2x4	80	225 MPa	159 MPa	215 MPa
25x2x4	200	241 MPa	207 MPa	227 MPa
50x2x4	400	247 MPa	218 MPa	237 MPa
100x2x4	800	242 MPa	223 MPa	231 MPa

Table 5.6: Von Mises stress: SolidFEM and Abaqus.

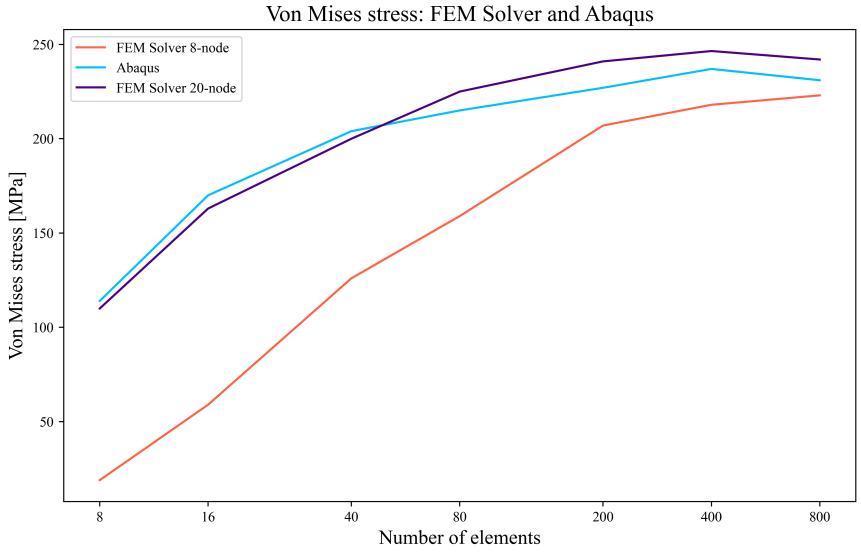


Figure 5.11: Study case 2, cantilever beam, stress: SolidFEM and Abaqus.

5.2.2 Arch beam

The other model analyzed is the arch beam. This introduces a more complex geometry to control if the solver can handle it. This case has a surface load, unlike the point load on the cantilever, applied to the whole upper surface of the arch. For supports, the left beam end is fixed in all directions in the bottom, while the right beam end is fixed in only the vertical and transverse directions. The geometry of the arch is shown in Table 5.7. This case uses the same material as the cantilever.

Width	Height	Arch length	Arch height	∑ Loads
100 mm	280 mm	3822 mm	1000 mm	-100 kN

Table 5.7: Model: Dimensions of the arch beam.

Grasshopper

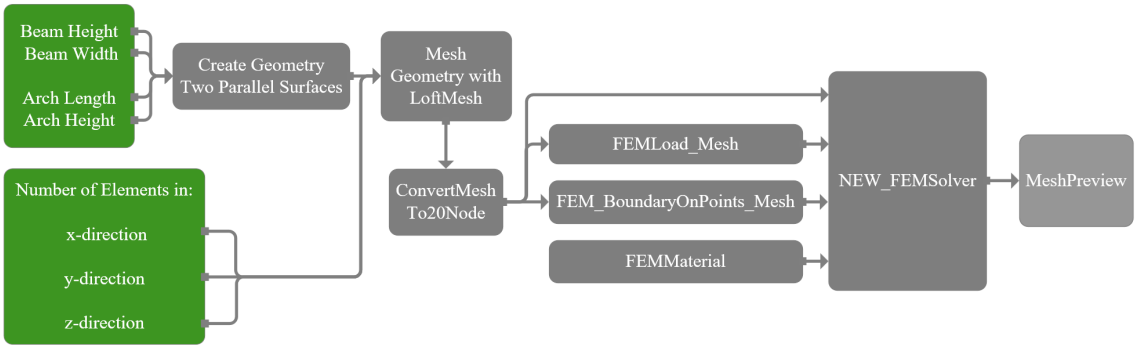


Figure 5.12: Flowchart of the Arched beam Grasshopper file.

A flowchart of the Grasshopper file is shown in Figure 5.12. The parametric model for the arch beam is built on the same basis as the cantilever beam. A set of input sliders defines the height and width of the beam, the length and height of the arch, and the number of elements in the x, y, and z directions for meshing. These inputs are used to create a simple arched curve that is expanded to two parallel surfaces. Using the **LoftMesh** component from the Simple FEM Solver plugin, a list of 8-node meshes was created. The **ConvertMeshTo20Node** component is then used to create 20-node meshes. These meshes are deconstructed and processed to extract the load surface on top of the arched beam, in addition to the support points at each end of the beam. The load vector is created using the **FEMLoadMesh** component. There are, as mentioned, different supports for the two beam ends. One of the beam supports is free in the longitudinal direction of the arch. These supports are created using the **FEMBoundaryOnPOintsMESH** component. The loads, supports and list of meshes are inputted in the **NEWFEMSolver** component, together with a material, to do the finite element analysis. The result is previewed with the **MeshPreview** component. Additionally, the maximum displacement and stresses are presented in a panel.

Abaqus

For this Abaqus model, the approach has been the same as for the cantilever. The geometry was created by creating the arch in 2D and extruding it along the width. A surface load was applied along the top of the beam. Supports were added for both beam ends, with the right end fixed only in the vertical and transverse directions. As for the cantilever model, the arch has been meshed by seeding three edges in three directions. In this way, the number of elements in each direction can be explicitly determined.

Results

As for the cantilever, the node where the results have been extracted from is shown in the figures with a yellow point. The results of the analysis are shown below, with Table 5.8 showing the displacements in the x direction (the longitudinal direction of the arch), Table 5.9 showing the displacement in the z direction (the vertical direction of the arch), and Table 5.10 showing the stresses. Figures 5.14 and 5.15 show displacement plots in the x and z directions with respect to the number of elements, while Figure 5.17 shows a stress plot with respect to the number of elements. Furthermore, the deformed geometry with colored displacements and stresses is shown in Figures 5.13 and 5.16.

Displacements

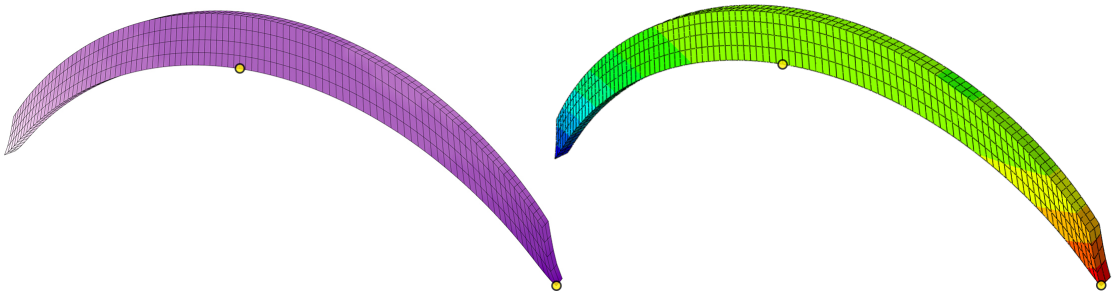


Figure 5.13: Case Study 2, arch beam model with displacements. SolidFEM (Hex 20) to the left and Abaqus (C3D20) to the right.

Mesh	No. Elements	SolidFEM 20-node	SolidFEM 8-node	Abaqus (C3D20)
10x2x4	80	3.051 mm	0.537 mm	2.858 mm
20x2x4	160	3.109 mm	0.854 mm	2.906 mm
50x2x4	400	3.216 mm	1.305 mm	2.999 mm
100x2x4	800	3.254 mm	1.400 mm	3.010 mm

Table 5.8: Displacement in the x direction: SolidFEM and Abaqus.

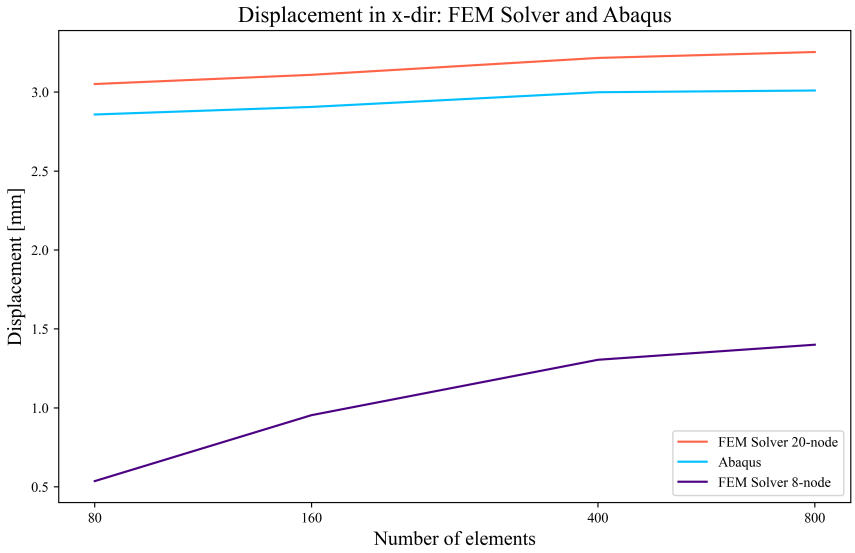


Figure 5.14: Displacement in x-direction: SolidFEM and Abaqus.

Mesh	No. Elements	SolidFEM 20-node	SolidFEM 8-node	Abaqus (C3D20)
10x2x4	80	-1.266 mm	-0.275 mm	-1.175 mm
20x2x4	160	-1.273 mm	-0.455 mm	-1.187 mm
50x2x4	400	-1.285 mm	-0.586 mm	-1.201 mm
100x2x4	800	-1.292 mm	-0.617 mm	-1.204 mm

Table 5.9: Displacement in z-direction: SolidFEM and Abaqus.

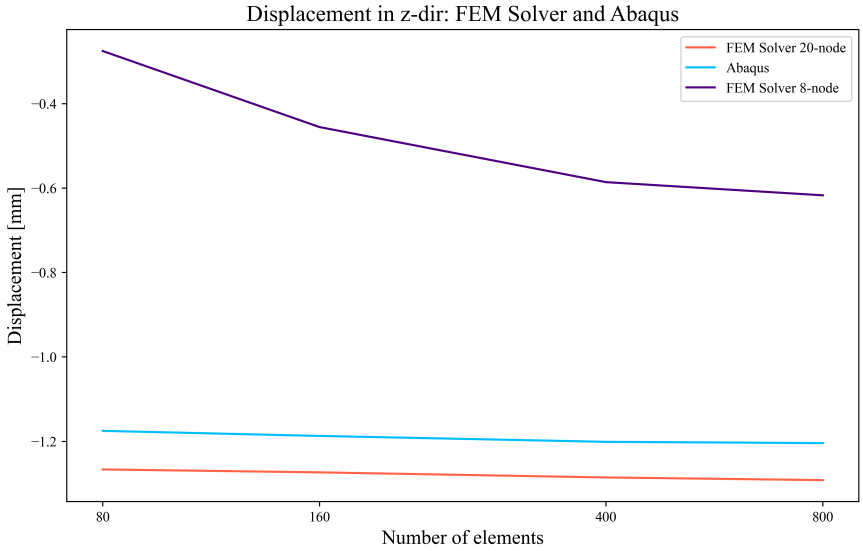


Figure 5.15: Displacement in z-direction: SolidFEM and Abaqus.

Stress

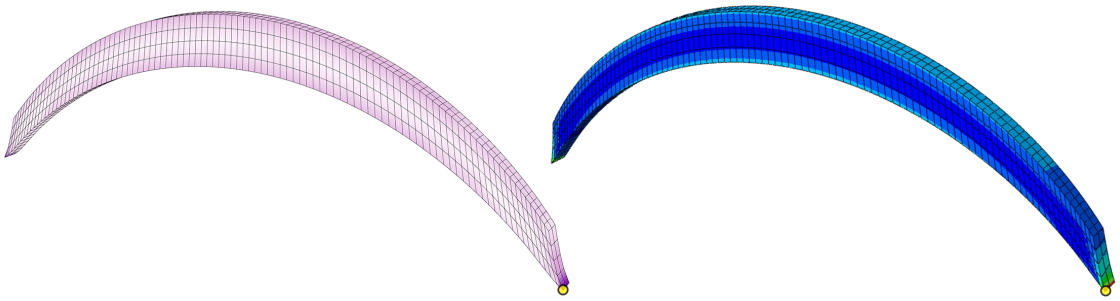


Figure 5.16: Case Study 2, arch beam model with stress. SolidFEM (Hex 20) to the left and Abaqus (C3D20) to the right.

Mesh	No. Elements	SolidFEM 20-node	Abaqus (C3D20)
10x2x4	80	29 MPa	28 MPa
20x2x4	160	37 MPa	42 MPa
50x2x4	400	81 MPa	79 MPa
100x2x4	800	124 MPa	125 MPa

Table 5.10: Von Mises stress: SolidFEM and Abaqus.

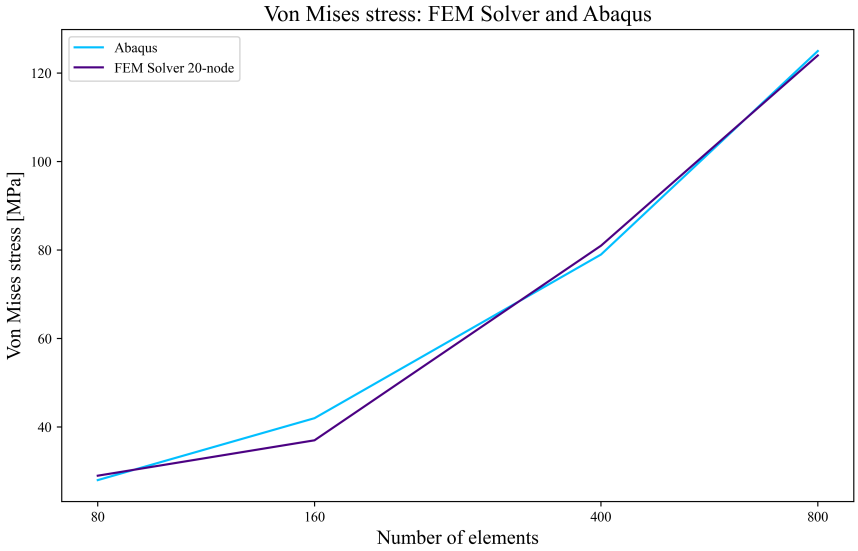


Figure 5.17: Case Study 2, arch beam, stress: SolidFEM and Abaqus.

5.2.3 Discussion

For the cantilever, the displacements from both SolidFEM with a 20-node element and Abaqus converge toward roughly the same value when the number of elements in the mesh increases. It is clear that the displacements coincide with the analytical solution in both Table 5.5 and Figure 5.9. The SolidFEM plugin with an 8-node element converges toward the same value, but at a much slower rate than the other solvers. The same trend is observable for the stress result shown in Figure 5.11.

The result of the comparison between SolidFEM and Abaqus was similar for both the arch beam and the cantilever. The deviation between the two is small for any number of elements. In both Figures 5.14 and 5.15, it can be seen that SolidFEM is consistently softer than Abaqus, as it has higher displacements in the x and z directions. For this curve geometry, the analysis with 8-node element in SolidFEM is clearly too stiff. This shows that when working with more complex geometries, higher-order shape functions improve the results. The stress results are very similar for SolidFEM and Abaqus. In Figure 5.17 it can be seen that the values do not converge toward one value but constantly increase as the number of elements increases. This comes down to the way the models were created, with supports only for the lower edge of the beam ends. With an increasing number of elements, this creates an increasingly high stress concentration for the support. Figure 5.16 also shows this effect, with high stress concentrations at the end of the beam.

Regarding the number of elements used in the different directions, many combinations were tried. To obtain a consistent result when increasing the number of elements, the division in the vertical and transverse directions was kept the same. It was observed that increasing the number of elements in the vertical direction had a large positive effect on the result, because the case is a beam in bending. Therefore, to achieve a consistent study on the effect of mesh refinement, the number of divisions in y and z direction was fixed.

In summary, the results of the SolidFEM solver are acceptable. For both the cantilever and the arch, the deviations from the analytical solution and established software such as Abaqus are of sufficient deviation. The solver calculates both displacements and stresses with good accuracy. It is also clear that, especially for curved geometry, the element with higher-order shape functions performs better. As seen in Figure 5.9, the 20-node element converges faster than the 8-node element, which confirms the theory between the 20- and 8-node element.

5.3 Introduction to Meshing with Machine Learning

The goal of the ML meshing part of this thesis is to develop a method that can be used to create an ML meshing component in the AAD environment. This component will replace one of the most challenging parts of the FEA performed in an AAD environment. The FEM solver component created in this thesis needs an adequate mesh as input. For now, the loft mesh component is used to create the mesh; this component works well, but is simple and has limitations when the geometry becomes complex. To perform an FEA with solid elements, we need a component that can create an adequate hexahedral mesh for complex geometry. Due to the complexity of the task and a desire to find the most optimal solution, the task has been restricted to finding a method to predict the location of the mesh vertices of a given object. The location of the mesh vertices is the most important information in a high-performing mesh. This performance is highly dependent on angular or curved-edge distortions, especially for lower-order elements. Our FEM solver uses the 20-node serendipity element. This element can model linear strain fields exactly as long as the element is rectangular. This capability is lost if the element is non-rectangular or if the edges are curved. The location of the mesh vertices is important to achieve a rectangular shape. If the object is optimally meshed with rectangular elements, a courser mesh can be used without loss of accuracy in the solution. This will simplify the problem and have a positive effect on the computational time of the analysis.

As an introduction, the task is first simplified to meshing of 2D surfaces, before moving on to the more complex implementation in three dimensions. Most of the research and methods developed are first implemented in 2D, before moving on to the 3D task.

The methods used and developed are inspired by Alexis Papagiannopoulos and Avellan, 2020. In this paper, a random contour is created and scaled to fit within a bounding box of grid points. The contour is then meshed using an existing meshing tool. The mesh vertices are then placed inside the grid, and a distance field (DF) is calculated. There are two different types of DF used in this thesis. The first DF calculates a scalar score for each point on the grid, representing the shortest distance to a mesh vertex. The other DF calculates a vector for each point; this vector represents the shortest vector to a mesh vertex. In the latter case, the DF is a vector field, while in the first case the DF is a scalar field representing the scalar value of the vectors. There is not a large difference in network performance between the two DFs, and both fields are used in this thesis. The DF is used as the target in our networks, whereas the input features that represent the geometry of the object vary with different network architectures. To make the following more intuitive and easier to see in the context of network architecture, it can be helpful to compare the grid points with scores to an image with pixels and RGB values. The difference is that the RGB values are replaced with a single scalar or a vector.

5.4 Case Study 3: 2D Meshing with Machine Learning

This case study starts with an explanation of how data are created in Python, and then we present two of the most successful network architectures that have been developed for this case study.

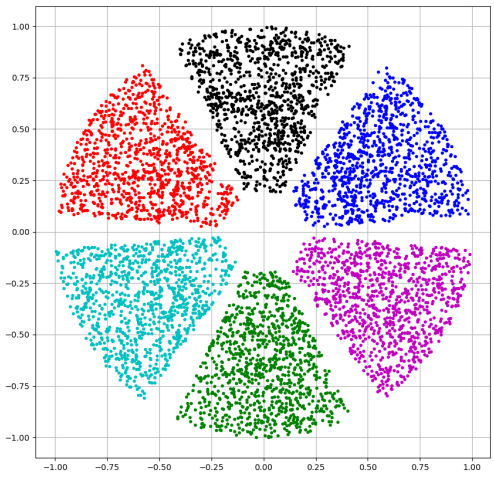
5.4.1 Data Generation

In the early stages of development, a custom function, **GetContour(n)**, was used to obtain a random polygon contour with n edges. In this function, the unit circle is divided into n zones, and polar coordinates are used to select a point in each zone. An example code can be seen in Listing 5.1. As can be seen in Figure 5.18a, some areas are excluded for point selection. This is done to avoid short edges; $\pm$ 5 radians are excluded between the zones. In addition, the **WeightedRandom** function excludes a circle area close to the center of the contour and has a weighting toward larger numbers to avoid sharp angles.

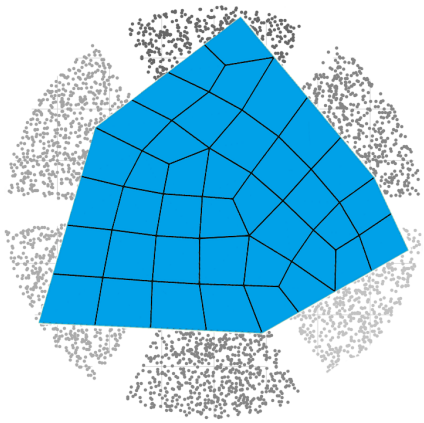
```
def GetContour(n):
    contour = []
    one_rad = pi/180
    rad = 2 pi/n
    theta = 0
    for i in range(n):
        a = random.uniform(theta+one_rad*5, theta + rad-one_rad*5)
        r = WeightedRandom()
        contour.append([r*cos(a),r*sin(a)]
        theta += rad
    return contour
```

Listing 5.1: Simplified code for the **GetCountour** definition.

The contour is normalized by translating the contour so that the mass center is located in origin. To have a unit diameter, the contour is scaled. This is important because we want the input contour to match the training contours when using the final model. In the final model, a contour of any size is scaled and translated, sent into the trained model before the prediction is scaled and translated back to its original size and returned. Figure 5.18a illustrates the creation of 1000 contours before normalization, a contour has one point from each zone.



(a) 1000 random hexagon contours.



(b) Hexagon meshed with Pygmsh.

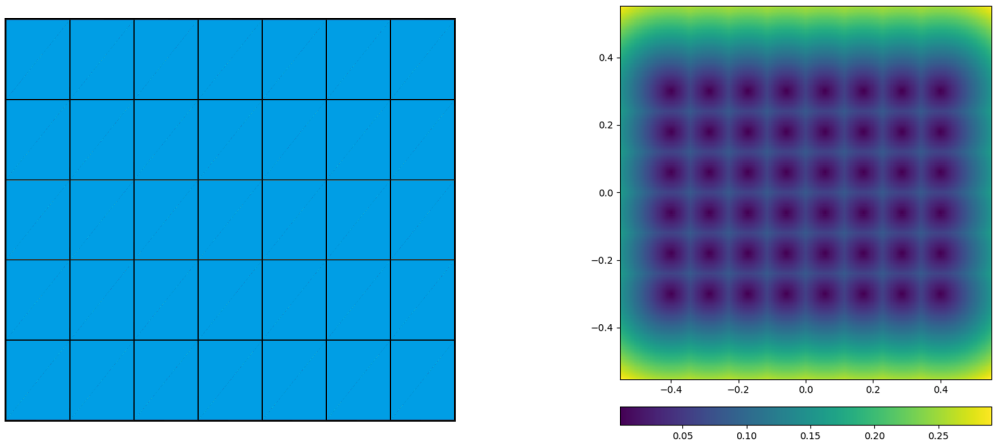
Figure 5.18: Data generation of Hexagons.

After creating a contour, the Pygmsh package is used to mesh the polygon. This mesh can be seen in 5.18b. The Pygmsh mesh vertices are used to create the DF used as a target in our networks. This approach looks fine and is similar to the method used in Alexis Papagiannopoulos and Avellan, 2020, but the performance of our networks never reached the level we expected or needed. After trying many different networks, fixes and techniques to improve performance, we started to investigate issues in the training data. To explore the trainability of our data, a GIF was created to investigate how the mesh looks for different contours. This GIF can be found in Table B.1.

The idea behind this GIF is to select two random contours from the training data, contour A and contour B. Then make 100 contours in between, where the first contour is A, the last contour is B, and the other contours represent a smooth transition from contour A to contour B. All the contours in this transition were meshed with the Pygmsh package. From this we expected to see a smooth transition in the meshes, as we saw for the contours, but this was not the case. For example, if we select contours 50 and 51 in this smooth transition, these contours look similar, but they can have totally different meshes. This jump in mesh configuration can be seen at some stages in the GIF and is not what we want in our training data. Two almost identical contours, or input to the network, can have two completely different meshes, or targets. In other words, the meshing output is discontinuous with respect to the input; therefore, any attempt to approximating the targets with a continuous function, as we are trying to do, will not yield acceptable results. To solve this problem, we need to find a meshing algorithm that creates an output that is continuous with respect to the input.

This issue was resolved using **StructuredGrid** provided by the Pyvista package in Python. This meshing algorithm has the desired mesh vertices as input. To simplify data generation in this case, the contour shape was changed to a four-sided polygon, a rectangle. The **GetContour** function is adjusted to create a rectangle where the side-length ratio varies. The code also includes an option to control the target number of elements in the mesh, and the generated mesh will have an element count as close to this number of elements as possible. The target number of elements is distributed in width and length so that the side length of each element is as equal as possible. The method developed here should work for any contour shape, as long as the generated mesh is continuous with respect to the input. After changing the meshing algorithm, we get the smooth transition we

wanted in the GIF. A link to the new GIF is found in Table B.1, while the new contour and mesh are illustrated in Figure 5.19a.



(a) Meshing with Pyvista.

(b) Distance field with scalar values.

Figure 5.19: Data generation where the target is continuous with respect to the output.

The vertices in the mesh are used to create the target, a DF. This field is illustrated in Figure 5.19b, where a color map is used to represent the scalar value of each point. As can be seen, the darker colors have a low score and represent local minima in the DF. The input features vary from network to network and are presented together with the different network architectures in Case Study 3 and 4.

5.4.2 Fully Connected Neural Network

For a fully connected network, the input features are the four points that describe the contour boundary, together with the grid points. Because the 2D contour is located in the xy-plane, the z coordinate is excluded. Therefore, each point has only two values, the x- and y-coordinates. The boundary box contains dim^2 = 65 536 grid points, corresponding to a grid with a dimension of 256 points in each direction. After concatenating the four contour points with the grid points, the input vector consists of 131 080 features. The target in our fully connected network is the DF with a scalar value for each point on the grid.

To find the optimal hyperparameters, a sweep run is completed in WandB, where the task was to minimize the validation loss. The best configuration of this run is presented in Table 5.11

Dropout	Layer sizes	Start learning rate	Validation loss
0.0	[512 1024 1024 512]	1e-03	0.01418

Table 5.11: Sweep run from WandB.

An illustration of the fully connected network can be seen in Figure 5.20 where $P_{1,x}$ represents the x-value of the first contour point and $P_{n,x}$ the last contour point. In this case $n = 4$. $GP_{1,x}$ the x coordinate for the first grid point and DIM^2 is the total number of points on the grid.

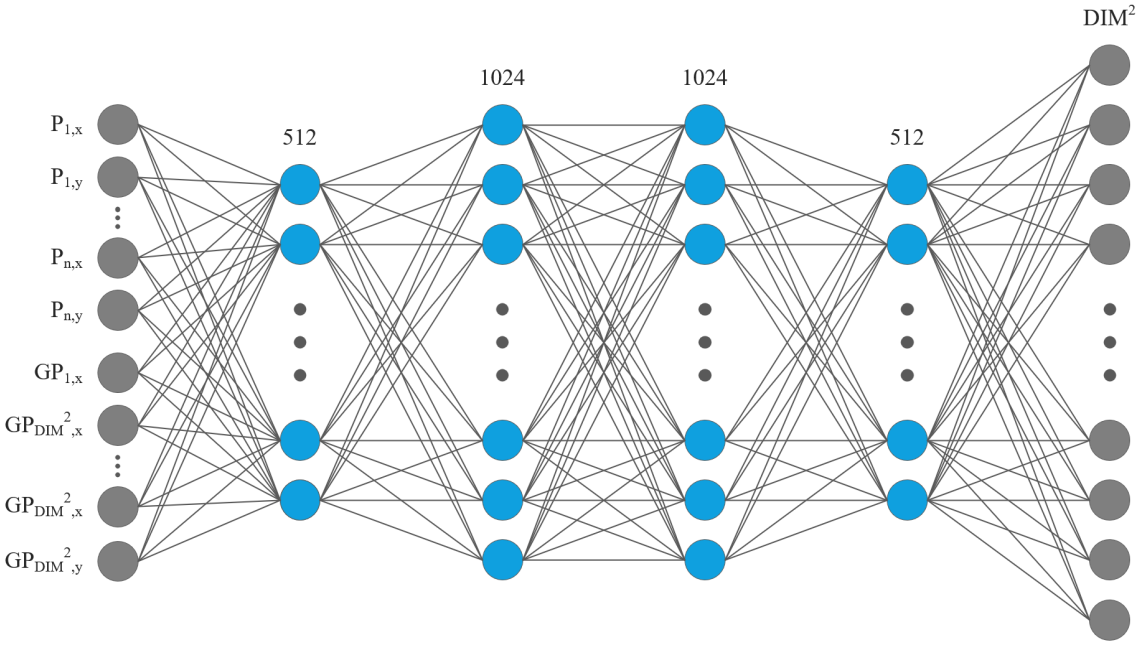


Figure 5.20: Illustration of the fully connected network.

Custom Loss Function

The L1 loss function is used to train our model, this function comes from torch.nn.functional and corresponds to the mean absolute error loss function described in Equation 2.29. If the reader recalls, we had some problems getting the performance we wanted when using the Pygmsh package. Before finding the reason for the low performance, a custom helper loss function, called DF loss, was developed. This loss function is meant as an addition to the L1 loss function and can be used for all networks in Case Study 3. The idea behind the function is classified within physics-informed neural networks because it uses physical properties of the DF. To present this idea, we need to introduce some theory.

The DF, referred to as f, in L_2-norm is differentiable almost everywhere and satisfies the Eikonal equation $||\nabla f(x,y)||_2 = 1$. More explicitly,

$$\left(\frac{\partial}{\partial x} f(x,y)\right)^2 + \left(\frac{\partial}{\partial y} f(x,y)\right)^2 = 1. \tag{5.2}$$

We can use this fact to formulate an additional loss for the output of a neural network. For a function g, the loss is

$$\mathscr{L}(g) := \iint (||\nabla g(x,y)|| - 1)^2 \, dxdy. \tag{5.3}$$

From Equation 5.3 it is intuitive to see that if g is a DF, then $\mathscr{L}(g) = 0$. To compute the partial derivatives, we use finite difference second order schemes (Brenner, 1960). The calculations used in the networks are illustrated in Listing 5.2.

```
# Second order schemes, forward difference
dx_kernel = torch.zeros(5,5)
dx_kernel[2] = torch.tensor([1/12, -2/3, 0, 2/3, -1/12])/step
dx_kernel = dx_kernel.view(1, 1, 5, 5)
dy_kernel = dx_kernel.transpose(2,3)

# Compute the finite partial derivatives
f_x = conv2d(DF, dx_kernel)
f_y = conv2d(DF, dy_kernel)
comb = torch.cat([f_y, f_x], 1)
f_norm = torch.norm(comb, dim=1)

custom_loss = ((f_norm-1)**2).mean()
```

Listing 5.2: Second order finite difference schemes.

The DF in Listing 5.2 is the prediction provided by the neural network, f_x and f_y represents the derivative with respect to x and y, and f_norm is the norm of the combination between the two. Figure 5.21 illustrates the calculations from a simple example problem. In the illustration, a DF with random points is created and used as input to the calculations.

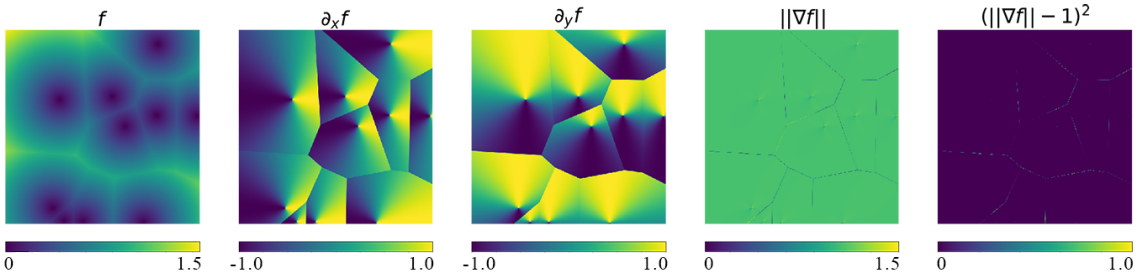


Figure 5.21: Illustration of partial derivatives with finite difference schemes.

The simple example in Figure 5.21 shows that the gradient norm is equal to one almost everywhere, that is, the loss is zero almost everywhere, except exactly at the points and along the boundaries. At the boundaries, the gradient jumps because the closest point suddenly changes. At these points, the derivative is not defined; these regions contribute a non-zero loss even for an exact DF.

Because the custom loss function is meant to be a helper loss function, the model is first trained with only the L1 loss function. This means that we are only training to optimize the L1 loss. We don't want the custom loss function to disturb the training at this point, we want it to fine tune the predictions after we all ready have a high performing model. A run with only the L1 loss is performed with early stopping included. Early stopping is included to stop the training after the model has converged; this saves a lot of time during training.

The trained model is then saved and used as a starting point for further training. At this point, the custom loss function is included, but multiplied by a scalar factor so that it is not too dominant in relation to the L1 loss. This is done to ensure that the helper function does not dominate the training. To find the most optimal scaling factor, a sweep run is performed. The results show that in this case, a factor of 1e-06 is the most optimal.

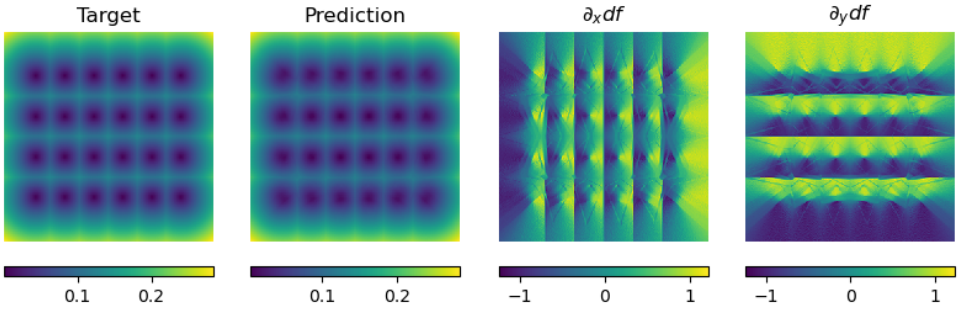
Results

When only the L1 loss is included in the optimizer and with early stopping, the validation loss converges after only eight minutes. At this point, we get the loss values listed in Table 5.12.

Valid loss	Valid loss L1	Valid loss DF
0.01415	0.01415	0.02642

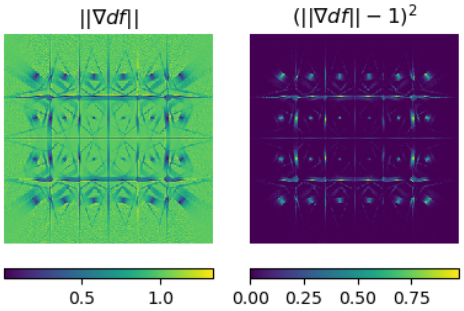
Table 5.12: Validation losses with only L1 in optimizer.

Note that the DF loss function is logged, but not included in the total loss that is used to train the model. An illustration of the prediction is shown in Figure 5.22.



(a) Target and prediction of DF.

(b) Partial derivatives of DF.



(c) Patrial derivative norm and DF loss.

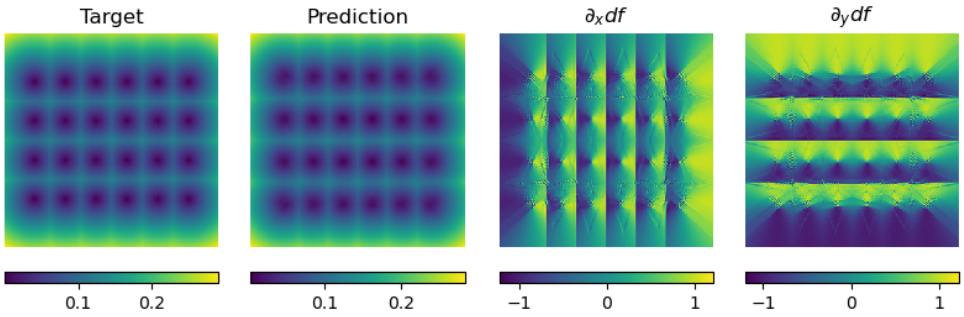
Figure 5.22: Results from training with only L1 loss function.

The network trained with the L1 loss function is saved and used as a starting point when including the DF loss function. Further training includes the L1 loss function and the DF loss function multiplied with the scaling factor 1e-03. The result can be seen in Table 5.13.

Validation loss	Validation loss L1	Validation loss DF
0.01411	0.01410	0.008954

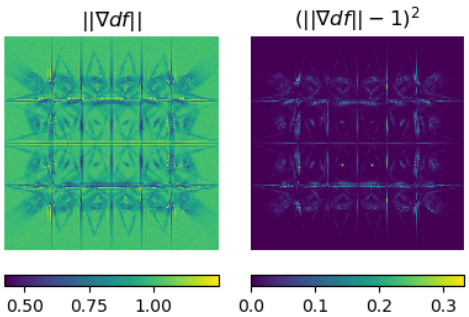
Table 5.13: Validation losses starting from the saved model, with DF loss included.

The image representation of the results from Table 5.13 is illustrated in Figure 5.23.



(a) Target and prediction of df.

(b) Partial derivatives of function df.



(c) Partial derivative and loss function.

Figure 5.23: Results from training with both loss functions.

Discussion

As can be seen from Figure 5.22, performance of the network trained with only L1 loss is decent. Although we do not include the custom loss function, the partial derivatives of the distance field look very similar to Figure 5.21, where the results are close to perfect. It is safe to say that the result from this network is good.

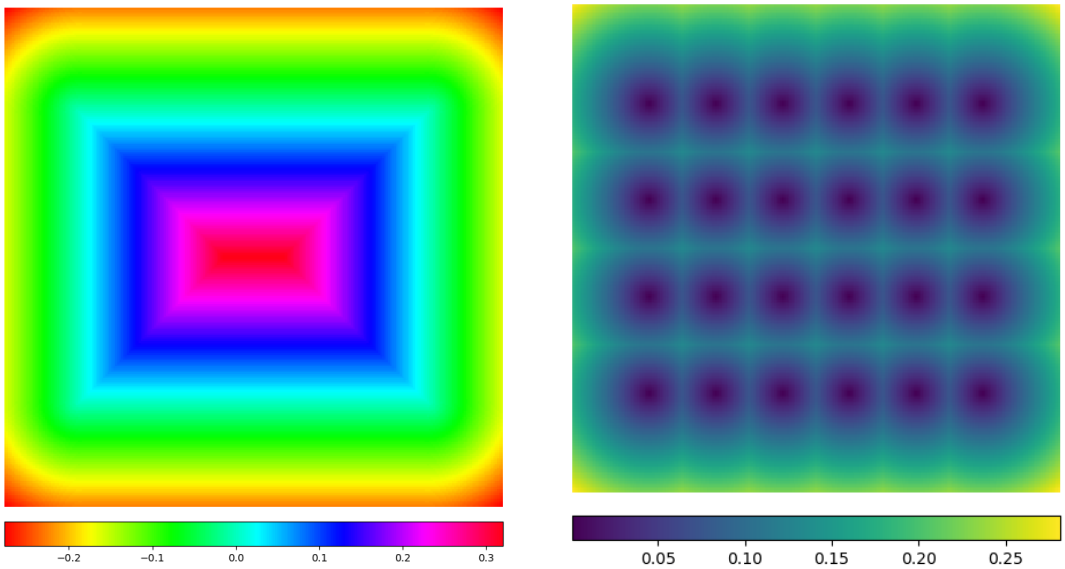
Table 5.13 shows the result when including both loss functions. As explained in the custom loss function section, there are some points in the distance field where the derivatives are not defined. These regions will introduce some error into the DF loss, even for a perfect estimate. Since we add the L1 and DF loss to get the total loss, the DF loss will also introduce an increase in the total loss of the model. Note that the DF loss is added to the total loss with the scaling factor included. The interesting question at this point is whether the loss of DF can help training so that the L1 loss decreases. Recall that it is the L1 loss that measures the error in the grid prediction; therefore, we want to reduce this loss. If the L1 loss does not decrease by including the DF loss function, then we have no reason to include it.

The result shows that including the DF loss function gives a slightly better performance; at least it does not hurt the grid score predicted by the L1 loss function. One theory to explain why the custom loss function does not improve the network more than it does is that the network already performs well. The reason for developing this custom loss function was to help the network learn at a stage where we had low performance. At this stage, the custom loss function did not help much, but, as mentioned, the low performance was due to discontinuity in the input-output relationship. Since the custom loss function slightly improves the predictions, it is reasonable to

believe that the theory used to develop the custom loss function is correct. It is also reasonable to believe that the custom loss function might have a greater impact on a network that does not perform as well. In simpler terms, the helper loss function cannot help improve the network if there is nothing to improve. Of course, the network can be further improved, but the results are decent and the location of the mesh vertices is easy to find from the grid prediction. It looks like the network is a good starting point to develop a complete mesh prediction algorithm.

5.4.3 Unet

This architecture is designed to take an image as input; in our case, we use a signed distance field (SDF) as input. An SDF is simply a DF as described before, but in this case the SDF describes the location of the contour and not the mesh vertices. The network architecture is transferable to our problem because the points in the grid are similar to, and can be compared to, the pixels in an image. The only difference is that the value of a point or pixel is a scalar, and not the RGB color representation. In an SDF, points outside the object have a negative value, points inside the object have a positive value, and points on the boundary have the value zero. An example of an SDF is illustrated in Figure 5.24a, while Figure 5.24b illustrates a DF. In these figures, the scalar values are represented with a color map.



(a) Input SDF of the contour.

(b) Target DF based on mesh vertices.

Figure 5.24: SDF and DF for the Unet architecture.

The input to the Unet is the grid points, where each point has a set of coordinates in addition to the scalar value provided by the SDF. In this case, the grid has a width and height of 256 points. Since the width and height are equal, the value is simply notated as dim. The SDF tensor has the shape [1,dim,dim]. The first element in the tensor is called the feature and represents the scalar value of a selected point, the second element is called the width and the last the height of the input. The grid tensor has the shape [3,dim,dim] where the feature represents the x, y and z coordinate of the point. The two inputs are concatenated into a tensor with four features, [4,dim,dim]. The output, or target, has the same shape as the SDF [1,dim,dim], where the feature represents the scalar value in the DF. Because this network is a bit more complex than the one explained in the method chapter, a list of the classes used in the network is included.

- **DoubleConv**: This class consists of two convolutional layers after another. It receives in-features, out-features, and optional mid-features as input. The mid-feature is only used in the **Up** class where we want to reduce the number of features. Note that features and channels are used interchangeably.

- **Down**: This class consists of a max pool that reduces width and the height of the grid, this is done to reduce memory allocated during training. It also contains the **DoubleConv** class that can change the feature size depending on the selected input. The input dimensions are set during the **init** function in the Neural network, as illustrated in Listing 5.3.
- **Up**: This class uses an up-sample class to increase the width and height of the grid. It also contains the **DoubleConv** class, which changes the size of the feature depending on the input. In this case, because of the skip connections, the mid-feature property is used in the middle of the two convolutional layers. In our network the mid feature has half the number of features compared to the in features.
- **OutConv**: This class is used at the end of the Unet and consists of a single convolutional layer. The output from this layer is the predicted DF.

The code in Listing 5.3 is included as an overview of the network. It illustrates the number of encoders, decoders, features, width and height of the grid at every step in the Unet. For this network we have not used a sweep run as we have for other networks, the layer sizes in the network are fixed as illustrated below. Hopefully, the architecture is well presented together with the class descriptions above, Figure 2.7 and the Unet theory.

```
class NeuralNet(nn.Module):
    def __init__(self, bilinear=False, device="cuda"):
        super(NeuralNet, self).__init__()
        self.bilinear = bilinear
        self.device = device

        # Unet
        self.inc = DoubleConv(4,64)
        self.d1 = Down(64, 128)
        self.d2 = Down(128, 256)
        self.d3 = Down(256, 512)
        self.d4 = Down(512, 1024)
        self.d5 = Down(1024,1024)

        self.u1 = Up(1024+1024, 1024)
        self.u2 = Up(1024+512, 512)
        self.u3 = Up(512+256, 256)
        self.u4 = Up(256+128, 128)
        self.u5 = Up(128+64,64)
        self.out = OutConv(64, 1)
```

```
def forward(self, SDF):      # [B,1,256,256]
    B = SDF.shape[0]         # Batch size
    dim = SDF.shape[-1]
    pts = PTS.unsqueeze(0).expand(B,-1,-1)          # [B,dim*dim,3]
    pts = pts.view(B,dim,dim,3).permute(0,3,1,2)    # [B,3,dim,dim]
    x = torch.cat((pts,SDF), dim=1)                 # [B,4,dim,dim]

    d = self.inc(x)          # [B,64,256,256]
    d1 = self.d1(d)          # [B,128,128,128]
    d2 = self.d2(d1)         # [B,256,64,64]
    d3 = self.d3(d2)         # [B,512,32,32]
    d4 = self.d4(d3)         # [B,1024,16,16]
    d5 = self.d5(d4)         # [B,1024,8,8]

    u1 = self.u1(d5,d4)      # [B,1024,16,16]
    u2 = self.u2(u1,d3)      # [B,512,32,32]
    u3 = self.u3(u2,d2)      # [B,256,64,64]
    u4 = self.u4(u3,d1)      # [B,128,128,128]
    u5 = self.u5(u4,d)       # [B,64,256,256]
    pred = self.out(u5)      # [B,1,256,256]
```

Listing 5.3: Unet architecture with sizes for every step.

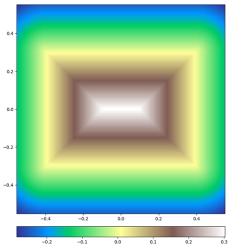
Results

With early stopping and learning rate scheduler, the model converged to the valid losses listed in Table 5.14. As before, the custom loss is logged but not included in the total validation loss.

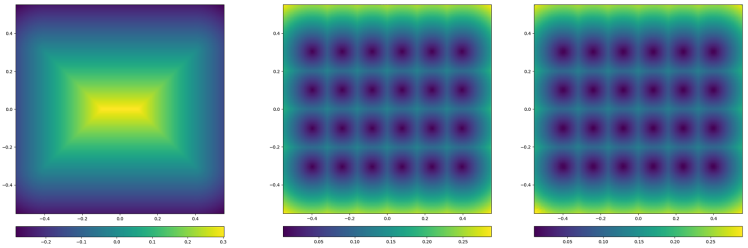
Validation loss	Validation loss L1	Validation loss DF
0.000936	0.000936	0.01241

Table 5.14: Validation losses with only L1 in optimizer.

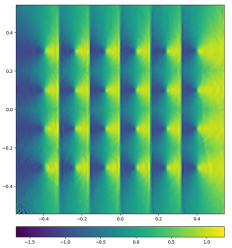
The input SDF is illustrated twice with different color maps in Figure 5.25a. Figure 5.25b illustrates the target DF on the left and the predicted field on the right. The partial derivatives are shown in Figure 5.25c and the last figure illustrates the combined derivative norm and the DF loss result.



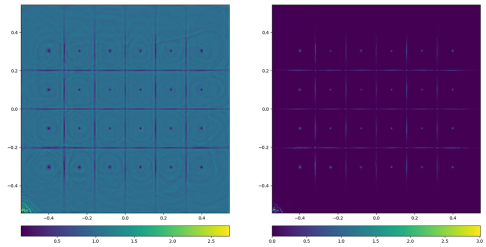
(a) SDF with terrain and viridis color map.



(b) Target and prediction of DF, f.



(c) Partial derivatives of DF, $\partial_x f$ and $\partial_y f$.



(d) Derivative, $||\nabla f||$ and DF loss, $(||\nabla f|| - 1)^2$.

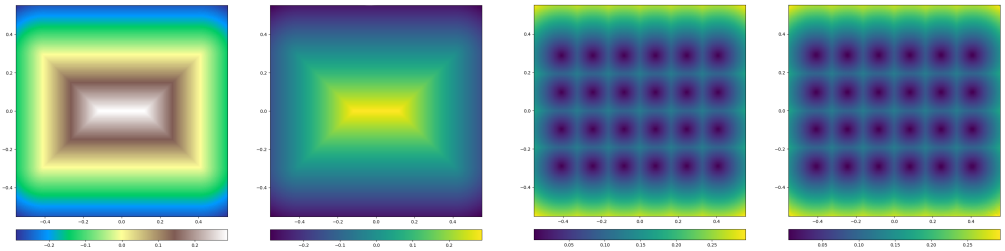
Figure 5.25: Results with only L1 loss function.

The trained model above is used as a starting point when including the DF loss function. To find the scaling factor, a sweep run is performed for values in the range of 1e-02 to 1e-09. The results give an optimal scaling factor of 1e-07. Table 5.15 lists the result when including both loss functions.

Validation loss	Validation loss L1	Validation loss DF
0.000930	0.000930	0.01004

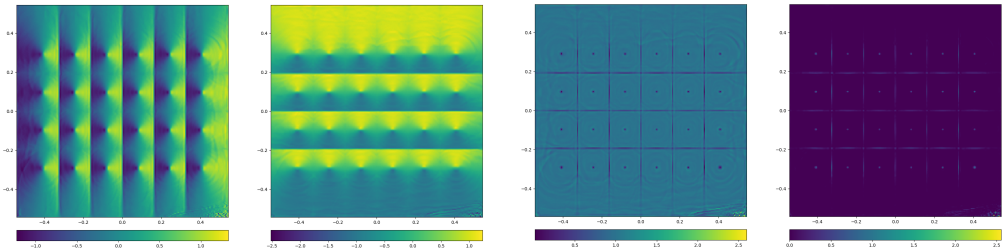
Table 5.15: Validation losses, L1 and Df loss functions.

The validation loss DF is added to the validation loss with the scaling factor, and the loss of the most interest is, as before, the validation loss L1. The custom loss function is worth including only if it reduces the valid loss L1 value. Figure 5.26 illustrates the results achieved by including the custom loss function.



(a) SDF with terrain and viridis color map.

(b) Target and prediction of DF, f.



(c) Partial derivatives of DF, $\partial_x f$ and $\partial_y f$.

(d) Derivative, $||\nabla f||$ and loss, $(||\nabla f|| - 1)^2$.

Figure 5.26: Results with both loss functions.

Discussion

As can be seen from the results, the Unet architecture with SDF as input performs even better than the fully connected network. By changing the input so that the Unet architecture can be used, we can improve the already decent predictions from the fully connected architecture. One disadvantage of using Unet architecture is that it uses more GPU memory. In the fully connected network, with the given depth, height of the network and a batch size of 64, approximately 60% of the GPU memory was allocated. This gives the possibility of increasing the grid point count and the accuracy of the predictions. In the Unet architecture, with the current number of encoders/decoders and batch size of 16 we allocate close to 100% of the GPU memory. Note that in both networks, the grid has a width and height of 256 points or pixels. The question is how many points are needed to achieve a good enough prediction of the mesh vertices. To answer this, we can ask another question: How accurate do we need the location of the vertices in the mesh to be? A mesh is already an approximation of the original geometry. Because of this, there already exists a difference from the true geometry and the mesh, even though it is small. In comparison, it is fair to say that the difference added by the mesh prediction error is negligible with the current resolution. We have 256 points evenly distributed over a length of 1.1, which gives a step size of 0.0043. With this step size and network performance, the error in the predicted location is very small. If, for some reason, we would like to increase the grid dimensions, there are some possibilities to do this. One option is to lower the batch size, another is to use different layer sizes in the encoder/decoder, or we could create an Unet with fewer encoders/decoders. For now, with the results we get, this is not something we want to explore. For the fully connected network, the DF loss function gives a small improvement in the predictions of the grid value.

5.5 Case Study 4: 3D Meshing with Machine Learning

Meshing with 3D is an extension of Case Study 3. The methods developed in 2D are also used here, only with an additional dimension included. The rectangle in Case Study 3 is extended to a box of a given height. Otherwise, the method with a DF as target is also used in this case study.

5.5.1 Data Generation

The **GetContour** function is adjusted to include a third dimension, the relation between width, length and height varies for each contour. As before, the number of elements is controllable, and the algorithm adjusts the number of elements in each direction so that the side lengths are as equal as possible. Figure 5.27 illustrates the meshed contour.

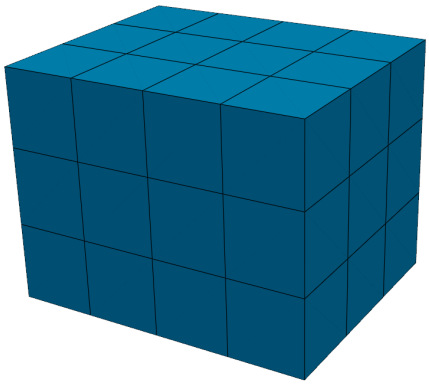
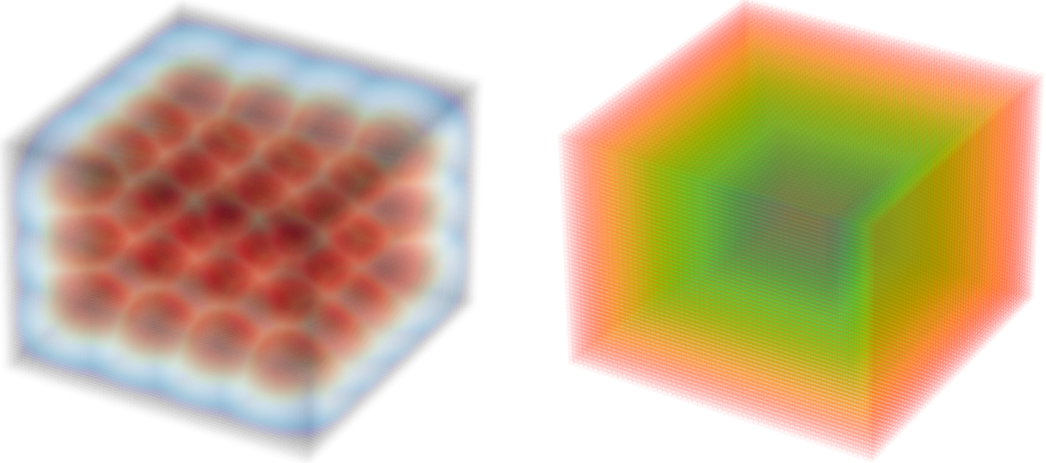


Figure 5.27: Illustration of 3D mesh.

The input to the networks used in this study is the contour points or an SDF. The target is, as before, a DF consisting of scalar values or a vector field. Figure 5.28a illustrates the DF of a grid with 80 points in each direction, while Figure 5.28b illustrates the SDF in three dimensions.



(a) Target DF in 3D. **(b)** Input SDF in 3D.

Figure 5.28: Illustration of DF and SDF in 3D.

5.5.2 Fully Connected Neural Network

This network is similar to the fully connected network created in Case Study 3. The difference is that the vector with the input features is larger. The input to the network is still the contour points together with the grid points. In the 2D case, the input vector had the total number of features calculated in Equation 5.4.

$$\text{Features} = \text{coords} \cdot \text{dim}^2 + \text{coords} \cdot \text{P} = 2 \cdot 256^2 + 2 \cdot 4 = 131\,080 \tag{5.4}$$

The same input feature equation can be calculated in 3D,

$$\text{Features} = \text{coords} \cdot \text{dim}^3 + \text{coords} \cdot \text{P} = 3 \cdot 40^3 + 3 \cdot 8 = 192\,024 \tag{5.5}$$

As can be seen in Equations 5.4 and 5.5, the transition from 2D to 3D gives an increase in the total number of input features. The only variable in the equations is the number of grid points or the number of points in each direction of the grid. This value is notated as dim in the equations. Even though dim is reduced from 256 to 40 we get an increase in input features. This may indicate that the fully connected architecture is not optimal for the 3D task. To answer this question, we need to find how many points are needed on the 3D grid to obtain an acceptable result. The number of input features must be balanced with the depth and width of the network. In the case where dim is set to be 40 in each direction, the depth of the network is four and the widths of the layers are [512,1024,1024,512], we allocate almost 100 % of the GPU memory. In this example, the GPU type used is NVIDIA GeForce RTX 3080.

In the following research, 40 points in each grid direction are used as default. This is done to ensure that there is available memory on the GPU to explore different layer depths and sizes, but also to get comparable results across the different network architectures. Note that some architectures are designed to use less memory. By restricting the grid dimension to 40 some of the advantages of these networks are not utilized, but the restriction is done to compare the results, and we always have the option to increase the grid dimensions after we are finished with the comparison.

A sweep run is completed to find an optimal combination of hyperparameters, the results of the best sweep run are listed in Table 5.16

Dropout	Layer sizes	Start learning rate	Validation loss
0.2	[512 512 512]	0.04	0.03305

Table 5.16: Sweep run from WandB.

Figure 5.29 illustrates the fully connected network with input features, depth and width, and the target vector.

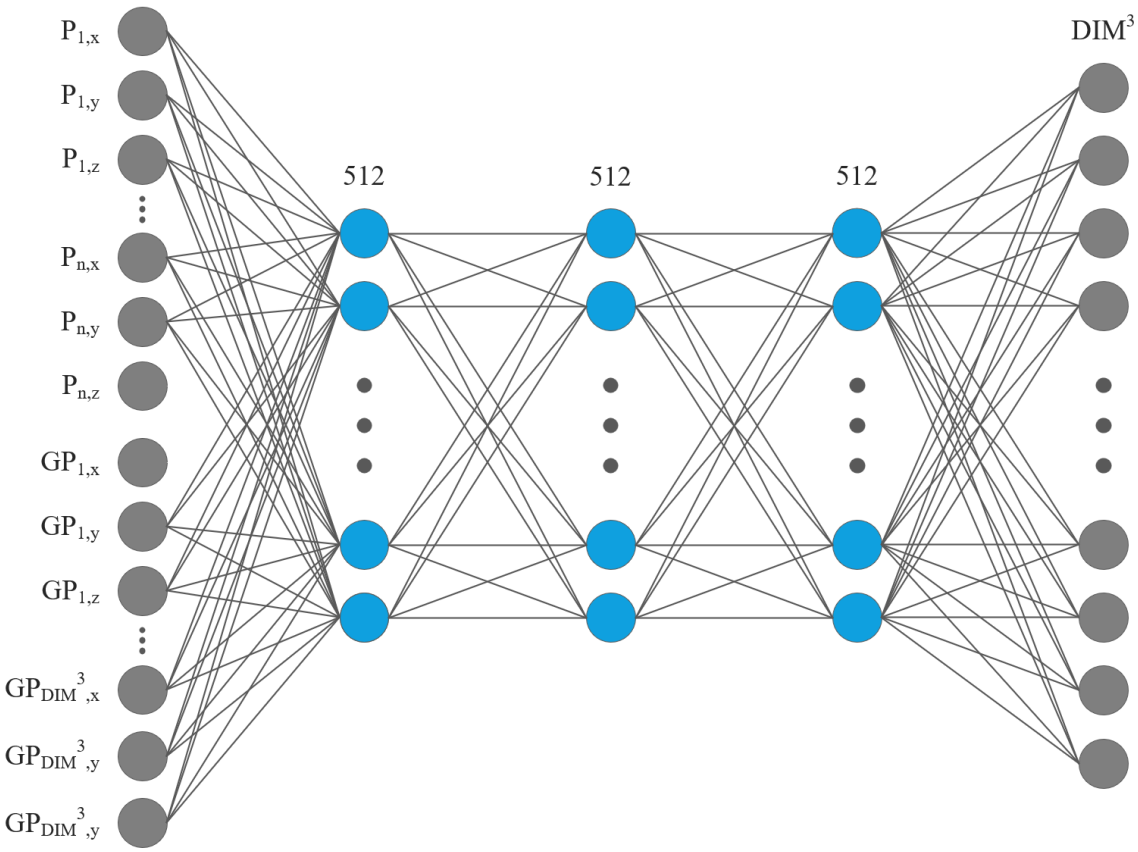


Figure 5.29: 3D fully connected neural network.

Results

The fully connected neural network can achieve the results listed in Table 5.17, this run is performed with the NVIDIA GeForce GTX 1080 GPU.

Validation loss	Dim	Layers	Batch size	Allocated GPU memory
0.03265	40	[512 512 512]	12	95%

Table 5.17: Validation loss.

Figure 5.30 shows the target and prediction of the grid scores illustrated with a color map. If comparing Figure 5.30 with the DF in Figure 5.28a, note that it is more difficult to create a good illustration of the DF when only using 40 grid points in each direction.

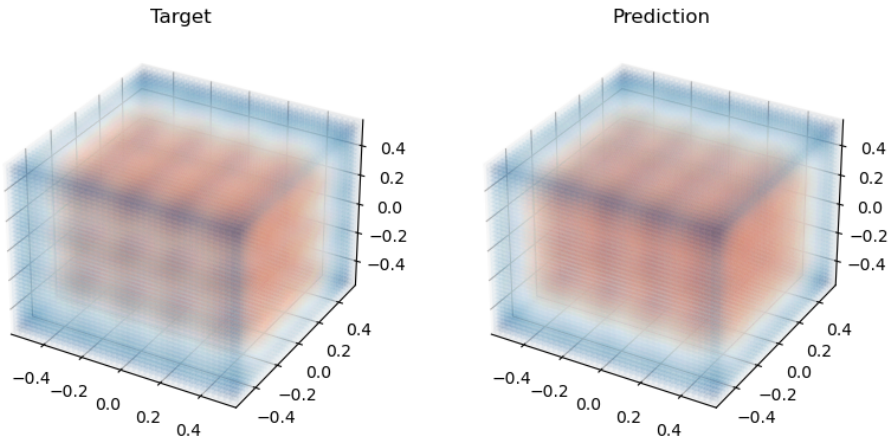


Figure 5.30: 3D fully connected neural network target and prediction.

Discussion

As can be seen in Tables 5.17 and 5.13 the prediction of the 3D grid is not at the same level as in the 2D case. One theory for why we are not able to achieve the same loss is that the 3D grid is a larger and more difficult problem. In the 3D case, it is hard to illustrate the predictions in a satisfying manner, especially in the case of few grid points. Therefore, in addition to the image in Figure 5.30, Figure 5.31 illustrates that we can locate the mesh vertices based on the predictions from the model. In this illustration, the trained model is used on a random example from the test dataset to obtain the predicted DF. Then, a code is applied that simply finds all the local minima in the DF. This not only illustrates that the predicted DF is accurate enough to find the vertices, but also illustrates that our trained and saved models are ready to be used on real examples.

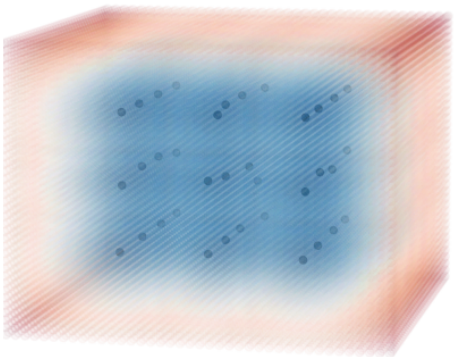


Figure 5.31: Location of mesh vertices based on prediction.

The only question at this point is whether the location of the vertices in Figure 5.31 is accurate enough. Recall that we are only using 40 points in each grid direction. The mesh vertices in Figure 5.31 are set to be the point with the lowest score in its neighborhood, a local minimum. As can be seen, the results are not quite as good as we would like, especially for the mesh vertices towards the center. An alternative approach is to select all points in a region around this local minimum. From these points, an interpolation between all values can be performed to find a more accurate location of the mesh vertices. This is something that would be preferable to do for the current results. The most intuitive approach to achieve a more accurate position of the mesh vertices is to increase the density of the 3D grid. This is not an option in this case due to the lack of available GPU memory.

5.5.3 Encoder + Fully Connected Neural Network

The main reason for developing this network is to reduce GPU memory allocated during training. By doing so, we get the possibility to increase the number of grid points, or use deeper and wider networks. The idea for this architecture is to use 3D convolutions together with 3D max pooling as an encoder for the input features. This can be compared to one of the encoder blocks in the **Unet** architecture presented in Case Study 3. Figure 5.32 illustrates the architecture of the combined networks together with the layer sizes decided by a Sweep run. The encoder has two layers with 32 neurons, while the fully connected layer has a depth of three layers, each with a width of 512 neurons.

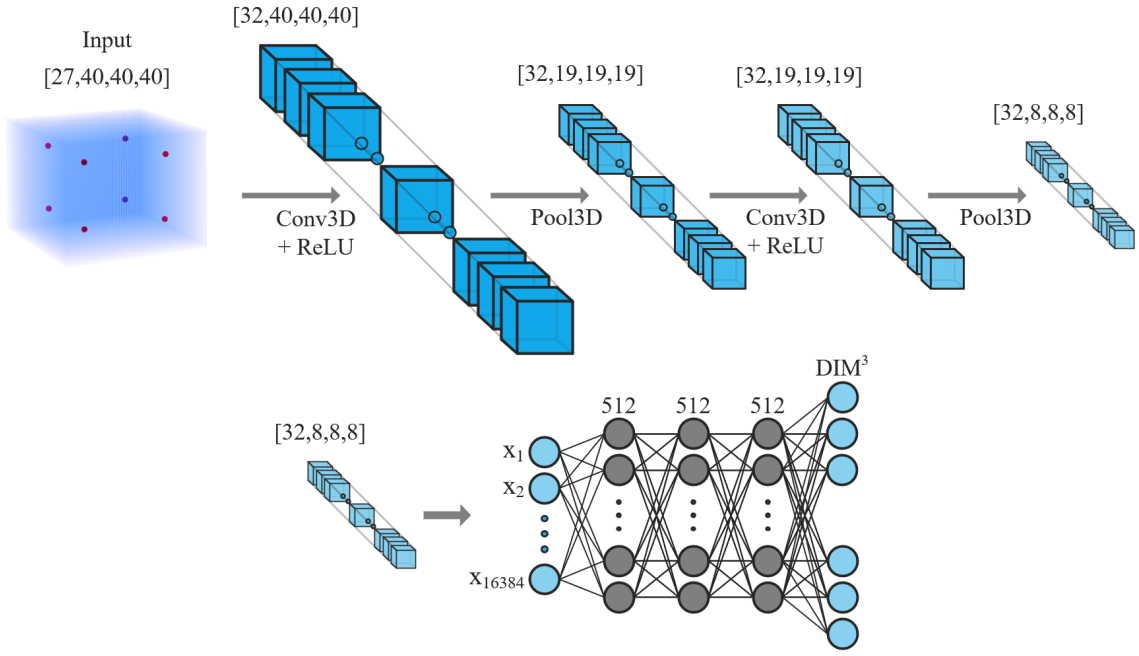


Figure 5.32: Encoder + fully connected neural network.

The input to this network is the contour and grid points, same as in the previous network. In the fully connected network described above, the input features are arranged in one long feature vector, while in this case the input has the original image or grid shape. The shape of the grid points are [dim,dim,dim,3], where 3 represents the x, y and z coordinate of the points. The shape is then rearranged so that the feature is the first element in the tensor; the shape is then [3,dim,dim,dim]. The original shape of the contour points is [8,3], where 8 is the number of points and 3 represents the x, y and z coordinates. These two numbers are flattened, three dimensions are added and repeated to obtain the tensor shape [24,dim,dim,dim]. This operation is applied to be able to concatenate the contour point representation with the grid points.

After concatenation, the encoder input has the shape [27,dim,dim,dim], where the features represent the contour points and the x, y and z values of a selected point on the grid. The output of the encoder has as many features as the last convolutional layer, in this case 32. The dim value is reduced from 40 to 8 in the encoder, giving the output shape [32,8,8,8]. From this feature representation, an input vector to the fully connected network is created. The feature vector inputted

to the fully connected network now has only 16 384 features. In comparison, the fully connected network without the encoder had a feature length of 192 024. The encoded features are sent into the fully connected network and return a prediction of the 3D DF.

Results

Training results for the network is listed in Table 5.18, training is carried out with the NIVIDA GeForce RTX 3080 GPU. Figure 5.33 illustrates the target and the prediction.

Validation loss	Dim	Conv. layers	Fully con. layers	Batch size	Allocated memory
0.02228	40	[32 32]	[512 512 512]	12	60%

Table 5.18: Validation losses from encoder + fully connected network.

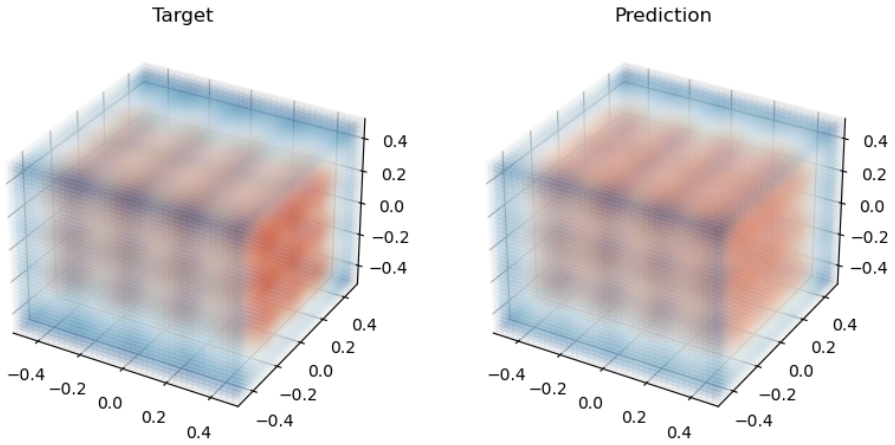


Figure 5.33: Target and prediction from encoder + fully connected neural network.

Figure 5.34 illustrates the mesh vertices predicted from the trained model. A vertex is located at each local minima in the predicted DF.

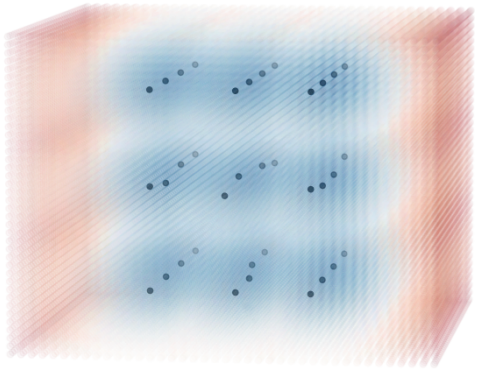


Figure 5.34: Location of mesh vertices based on prediction.

Discussion

By including the encoder before sending the features into the fully connected network, we were able to reduce the allocated GPU memory from close to 100 % to about 60 %. The results in Figure 5.34 are based on a grid with only 40 points in each direction, using this architecture we have the possibility to use a denser grid. We also see a decrease in loss by encoding the features before sending them through the fully connected network. The location of the vertices in Figure 5.34 is not quite at the level we want to achieve, but the results are improved compared to Figure 5.31. As discussed above, there are at least two ways to improve the predicted location of the mesh vertices. Interpolate a region around the local minima or increase grid density.

Since we can increase the density of the grid for this network, we conducted a study on its effect. In the study, new data with different grid dimensions are created, and sweeps are performed to find the optimal configuration of hyperparameters for each case. The results from this study are listed in Table 5.19.

Dim	Conv. layers	Fully con. layers	Allocated memory	Loss
40	[32 32]	[512 512 512]	60 %	0.02228
50	[32 32]	[128 128 128]	50 %	0.02142
80	[32 32 32]	[128 128 128]	99 %	0.02225

Table 5.19: Study of the effect of different grid sizes.

Figure 5.35 illustrates the target and prediction of a grid with 80 points in each direction. Note that a layer is added to the encoder block in this case, this is done so that we were able to have three layers with a width of 128 neurons in the fully connected part without exceeding the GPU memory capacity.

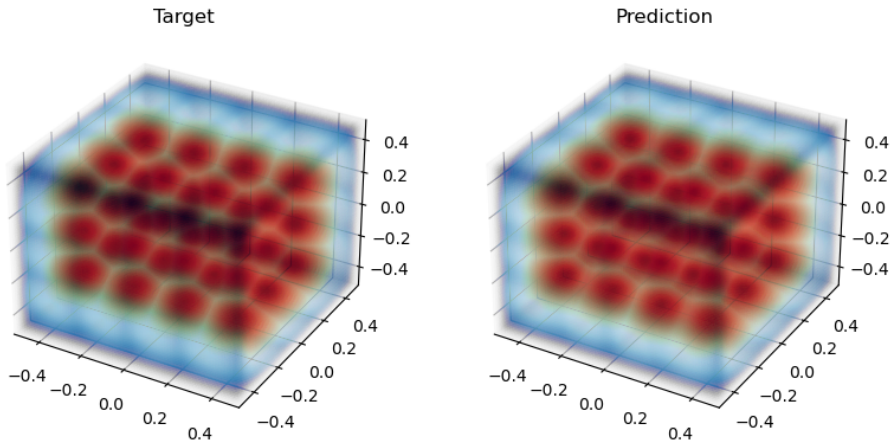


Figure 5.35: Target and prediction with 80 grid points in each direction.

Figure 5.36 illustrates the predicted mesh vertices in a grid with 80 points in each direction. When comparing Figure 5.36 with Figure 5.34, it is clear that increasing the number of points on the grid has a positive effect on the predicted location of the mesh vertices. The predictions of vertices in a

grid with dimensions of 80 points are very close to predicting the correct location. The predictions from this network are well within the accuracy needed to create an acceptable mesh.

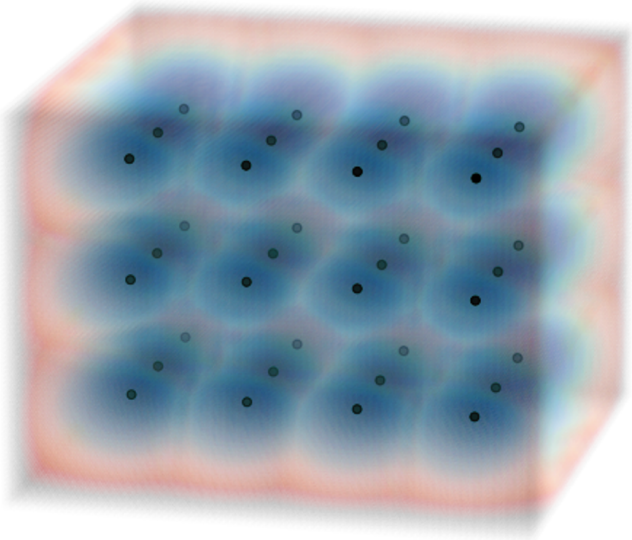


Figure 5.36: Location of mesh vertices based on prediction.

Due to the available time, we have not optimized the grid density and network depth/width relationship; this is something that should be fine-tuned if using this architecture. Some of the Sweep runs, with lower network width, give a loss almost as low as the case with higher network width. Therefore, we believe that it is possible to create an even denser grid and get an even more accurate location of the mesh vertices. However, it is clear that encoding the input features before sending the information through the fully connected network is preferable to a fully connected network alone. After the grid dimension study, we also have a strong indication that, if optimizing the network, the predictions can be good and well within the accuracy needed to predict the location of the mesh vertices.

5.5.4 3D Unet

This last network is based on the **Unet** architecture. The **Unet** was originally created for 2D images, not for 3D grids, or 3D images if you want. The main idea in this network is to replace the 2D convolutions, 2D max pooling, and 2D up-scaling with 3D versions. To free up some memory on the GPU, we have replaced the **DoubleConv** class with a **ResBottleneckBlock** class. A Bottleneck Residual Block is simply a block that utilizes 1x1x1 convolutions to create a bottleneck. The bottleneck reduces the number of parameters and matrix multiplications. **ResBottleneckBlock** makes the block as thin as possible to increase depth and have less parameters (Kaiming He & Sun, 2015). After implementing the adjustments, we get the network illustrated in Listing 5.4. Note the dimensions written in the comments.

```
class NeuralNet(nn.Module):
    def __init__(self, bilinear=False, device="cuda"):
        super(NeuralNet, self).__init__()
        self.bilinear = bilinear
        self.device = device

        # Unet
        self.inc = ResBottleneckBlock(4,64)
        self.d1 = Down(64, 128)
        self.d2 = Down(128, 256)
        self.d3 = Down(256, 512)
        self.d4 = Down(512, 1024)
        self.d5 = Down(1024,1024)

        self.u1 = Up(1024+1024, 1024)
        self.u2 = Up(1024+512, 512)
        self.u3 = Up(512+256, 256)
        self.u4 = Up(256+128, 128)
        self.u5 = Up(128+64,64)
        self.out = OutConv(64, 1)

    def forward(self, SDF):                          # [B,64,64,64]
        B = SDF.shape[0]                             # Batch size
        dim = SDF.shape[-1]
        SDF = SDF.view(B,1,dim,dim,dim)              # [B,1,dim,dim,dim]
        pts = PTS.unsqueeze(0).expand(B,-1,-1)       # [B,dim*dim*dim,3]
        pts = pts.view(B,dim,dim,dim,3)              # [B,dim,dim,dim,3]
        pts = pts.permute(0,4,1,2,3)                 # [B,3,dim,dim,dim]
        x = torch.cat((pts,SDF), dim=1)              # [B,4,dim,dim,dim]
```

```
d = self.inc(x)          # [B,64,64,64,64]
d1 = self.d1(d)          # [B,128,32,32,32]
d2 = self.d2(d1)         # [B,256,16,16,16]
d3 = self.d3(d2)         # [B,512,8,8,8]
d4 = self.d4(d3)         # [B,1024,4,4,4]
d5 = self.d5(d4)         # [B,1024,2,2,2]

u1 = self.u1(d5,d4)      # [B,1024,4,4,4]
u2 = self.u2(u1,d3)      # [B,512,8,8,8]
u3 = self.u3(u2,d2)      # [B,256,16,16,16]
u4 = self.u4(u3,d1)      # [B,128,32,32,32]
u5 = self.u5(u4,d)       # [B,64,64,64,64]
pred = self.out(u5)      # [B,1,64,64,64]
```

Listing 5.4: 3D Unet architecture with sizes for every step.

Results

The training results achieved for 3D Unet are listed in Table 5.20, and training is performed with the NVIDIA GeForce RTX 3080 GPU.

Validation loss	Dim	Batch size	Allocated GPU memory
0.02511	64	8	98 %

Table 5.20: Unet training results.

Figure 5.37 illustrates the predicted distance field as a color map. Note that this figure is easier to illustrate because it has 64 grid points in each direction and not 40 grid points, as is the case for figures such as 5.33. Because of this, Figure 5.37 may give a false impression compared to figures with a dimension of 40 points in each direction.

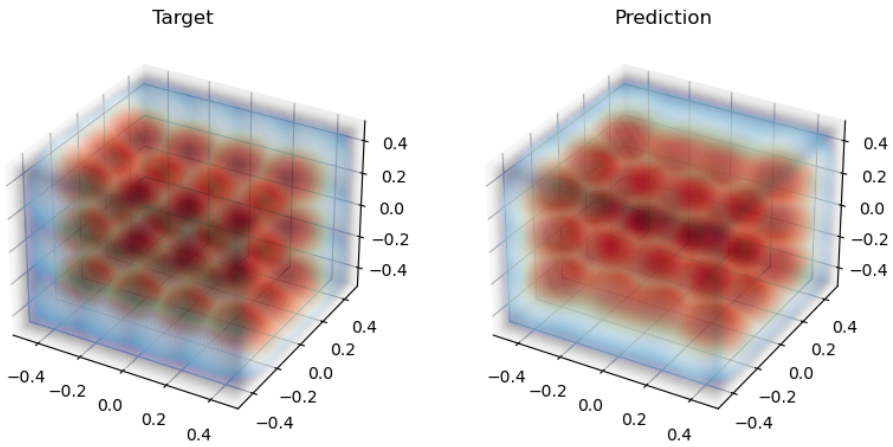


Figure 5.37: Unet target and predicted distance field.

Figure 5.38 illustrated the location of the mesh vertices based on the prediction provided by the **Unet** model.

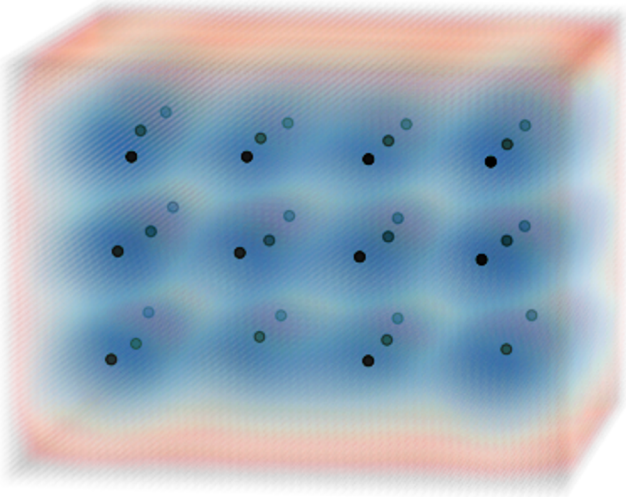


Figure 5.38: Predicted location of mesh vertices.

Discussion

In the case of **3D Unet** we had to change the **DoubleConv** class with a **ResBottleneckBlock** class to obtain predictions that look close to the target. Even with this change, we do not get predictions on the same level as for the encoder + fully connected network. This is clear from Figure 5.38, where we miss two of the vertices. As can be seen in Figure 5.37 the model struggles with predictions of nodes on the mid side. This is a natural result of a lower performing model. Our data usually have three or four vertices in each direction. This variable of mesh vertices makes the predictions harder, and that is why these mid-side vertices look kind of smudged. The model is not sure if it should predict three or four vertices, so it predicts something in between. A solution to this problem is to increase the number of points in the grid, but a large increase in grid points is not possible due to lack of GPU memory. The Unet performed very well in the 2D case; if we could increase the number of grid points, we believe that we could achieve a similar result for the 3D case.

It is worth noting that we have not spent a lot of time optimizing this architecture. It is possible that some fixes or changes can help to further improve the predictions, but as it is now, this network is not optimal. Some changes that can be done, so that we are able to increase the number of grid points, are to use fewer encoder/decoder blocks, change layer sizes in the convolutions, or run the network on a GPU with more memory. Due to the high performance of the **2D Unet** we believe that using some more time to develop this architecture may be worth exploring.

5.6 Case Study 5: FEA of Cantilever Beam with ML

In this case study, the prediction of displacement and stress with ML for a simple cantilever beam is investigated. The dataset is created in Grasshopper using the SolidFEM plugin for the FEA. There are two files in the datasets, one file containing input features and one file containing targets. The input is used to represent the geometry and the information needed for the FEM analysis. For this case, the only information needed to represent the beam is a vector, since the cross-section is the same for all cases. Also, a scalar is needed to represent the magnitude of the load. The target file contains displacements and stresses for all vertices of the mesh, in addition to the coordinates of the corresponding point. To train the model, the framework described in Section 4.3 is used. Much of the work has gone into optimizing the network to obtain the best possible results. This section will first describe the method for creating the dataset, then describe the final version of the training loop before finally describing the development process. To create the final working network, many different approaches have been tried. The most important of these will be described to show the progress that led to the final version.

5.6.1 Dataset creation in Grasshopper

The Grasshopper file for data creation consists of four main parts: Create Geometry, Mesh Geometry, FEA and Create dataset. To create a lot of different cases, the entire Grasshopper code is looped by a custom C# component. This component has a number slider as input and will loop through all values possible by the slider. That is, if the slider has a max value of 100, and a step of 1, the component will loop 100 times. For each loop, a new geometry is created; this geometry is meshed and analyzed, and the results of the analysis are written to a file together with the corresponding inputs. A flowchart of the Grasshopper file is shown in Figure 5.39.

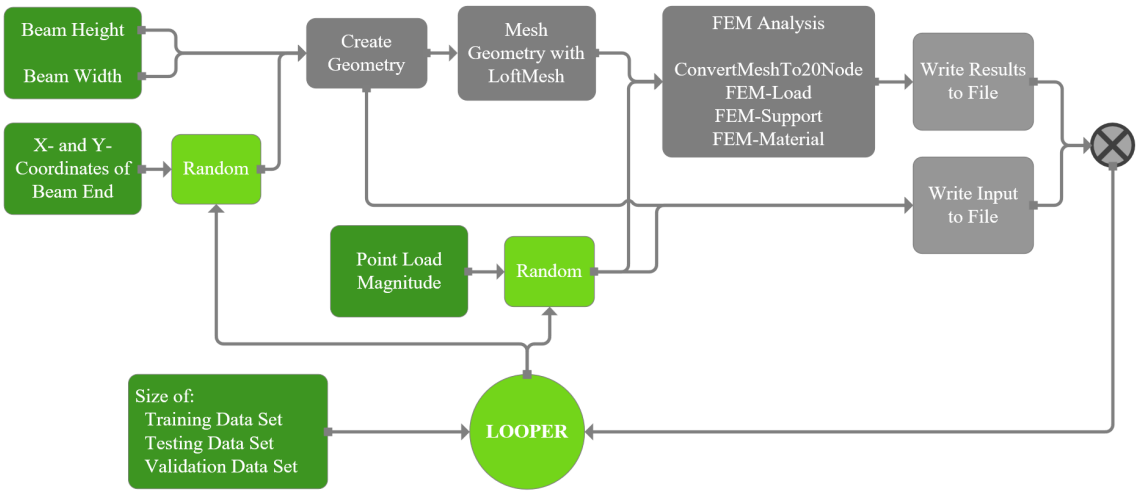


Figure 5.39: Flowchart of the data creation in Grasshopper.

For geometry creation, a random number between 1000 and 2000 is used for the x and y coordinates for the end of the beam. This creates a beam in the first quadrant of a unit circle with a length between 1414 and 2828 mm. The number of iterations in the loop is used as a seed to ensure that a new random number is chosen for each iteration. From this length, a rectangle with a height of

300 mm and a width of 100 mm is created and extruded along the beam. This creates two surfaces, one for the top and one for the bottom. The geometry of the cantilever is listed in Table 5.21.

Width	Height	Length	∑ Point Loads in Z direction
100 mm	300 mm	1414 to 2828 mm	-100 to 100 kN

Table 5.21: Model: Cantilever beam.

These two surfaces are inputted into the **Loft Mesh** component described in Section 4.1.1. This creates an 8-node mesh, which is inputted in the **ConvertMeshTo20Node** component, to create a 20-node mesh. This mesh is then used in the analysis part of the code. Support and load points are extracted from the mesh and used to create supports and loads. The load is a point load at the end of the beam, which varies between plus and minus 100 kN. This value is randomly chosen in the same way as the coordinates for the beam. All of this is inputted into the **NEWFEMSolver** component, together with the material. In this case the material is steel, with the properties listed in Table 5.22.

Young's Modulus	Poisson's ratio	Yielding stress	Material weight
210000 MPa	0.3	355 MPa	7850 kg/m^3

Table 5.22: Properties of the steel material.

The last part of the script is to sort the results into an easily readable form. This is done by concatenating displacements in the x, y and z directions, the nodal von Mises stress, and the corresponding x, y, and z coordinates. This list of results is then used in the dataset creation part of the Grasshopper code.

To write the results from the analysis to a .txt file, a custom Python script is used. The code for this illustrated in Listing 5.5. This component uses FileName and the name for Folder, as well as the lines to be written. Also, it has a stopper input which is used to create different cases for the training, testing, and validation sets. The total number of sets is determined by the length of the slider that is inputted in the loop component. This number has to be divided into train, test, and validation. If 10 000 sets are to be created, 9000 of them could be for training, then 500 would for testing, and the last 500 for validation. When doing it this way, duplicate sets are avoided. In addition to writing the results to a file, the input used to create these results is written to another file in the same way. This input includes a vector representing the beam and the point load used at the end of the beam.

```
if stopper == True:
    exit()                          # Stops the script if stopper is true
File = Folder + "/" + FileName      # Create folder and filename

# Create directory
dirName = Folder
if not os.path.exists(dirName):
    os.mkdir(dirName)
```

```
FilePath = open(File, "w")          # Open text File

if os.stat(File).st_size == 0:
    for line in lines:              # Iterate through lines
        FilePath.write(line+"\n")   # Write separate lines
else:
    for line in lines:              # Iterate through lines
        FilePath.write(line+"\n")   # Write separate lines

FilePath.close()                    # Close the File
```

Listing 5.5: Listing of the Python script that writes data to a file in Grasshopper.

5.6.2 Postprocessing of Datasets

Because the data is created in Grasshopper and not in Python, some adjustments are needed to the **dataset.py** file. To make training easier and the input data more consistent, the beams are rotated to align with the x-axis. This is done by calculating the angle between the beams neutral axis and the x-axis, before multiplying the beam with a rotation matrix. Furthermore, some of the input features needs to be scaled to avoid large gradients that can cause a problem when training the network. The scaler applied in this case is the StandardScaler provided by the *scikit-learn* package, this scaler standarizes the values around zero. The scaler is applied to both the forces and the points describing the beam. This means that we have two scalers in this network, one for the forces and one for the coordinates. The **dataset.py** returns the scaled force, scaled and rotated coordinates, original coordinates, target stress, target displacement and the rotated vector describing the beam. Original coordinates are included to be able to plot an image representation to WandB.

Fully Connected Neural Network

In this Case Study a fully connected architecture is applied as the neural network. The input layer has the shape [P, 7], where P represents all mesh points from the FEM analysis. Each of these points has seven features, the scaled force, rotated and scaled x, y and z coordinates in addition to the vector describing the beam. The input layer is then connected to three hidden layers with the width listed in the sweep run Table 5.23. The output layer has the shape [P, 4], where each point P is mapped to a predicted stress value and displacement in the x, y and z directions. The output is divided into two predictions, [P,3] for the displacements and [P,1] for the stress values. From the predictions, three losses are logged at WandB. The first loss is the L1 loss for the displacement, the second is the L1 loss of the stresses and the last loss is the combination of both losses. This is done to investigate how the model performs on the predicted displacements and stresses separately. In addition to the losses, a graphical representation of the predicted result is logged. The representation is illustrated in Figure 5.40, where the displacements are scaled, so that they are visible, and added to the original coordinates while the stresses are illustrated with a color map.

5.6.3 Result

To decide the best network configuration, including network depth and width, hyperparameters, start learning rate and dropout, a sweep run with the task of minimizing the validation loss is performed. The result of this sweep is listed in Table 5.23.

Dropout	Layer sizes	Start learning rate	Validation loss
0.0	[256 512 256]	0.007	0.3072

Table 5.23: Sweep run from WandB.

Table 5.24 lists the results achieved from a long run with the optimal configuration.

Validation displacement loss	Validation stress loss	Validation loss
0.226	0.042	0.267

Table 5.24: Resulting validation loss.

Figure 5.40 gives an illustration of the displacements and stresses. This figure illustrates the target and predictions of a random example from the test set.

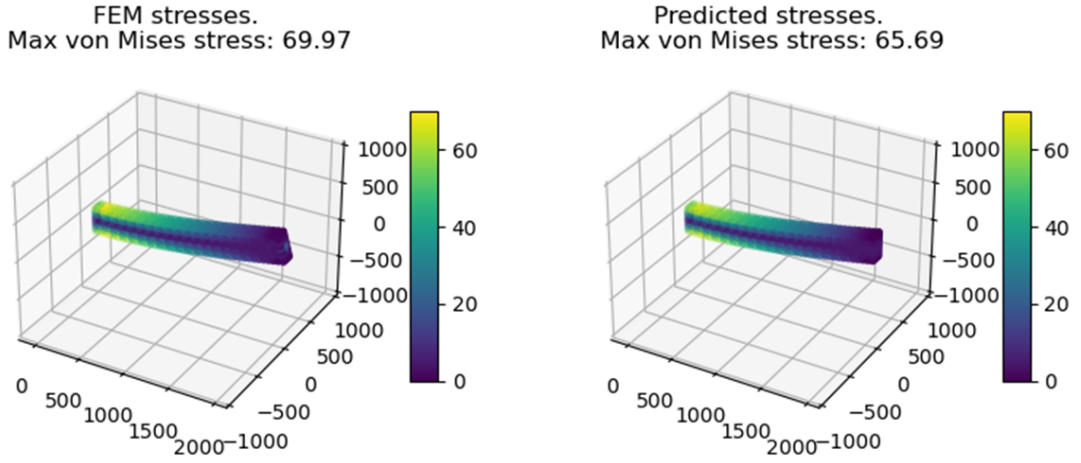


Figure 5.40: Target and predicted displacement.

To visualize the training process, Figure 5.41 illustrates the target along with the predictions at different steps or iterations.

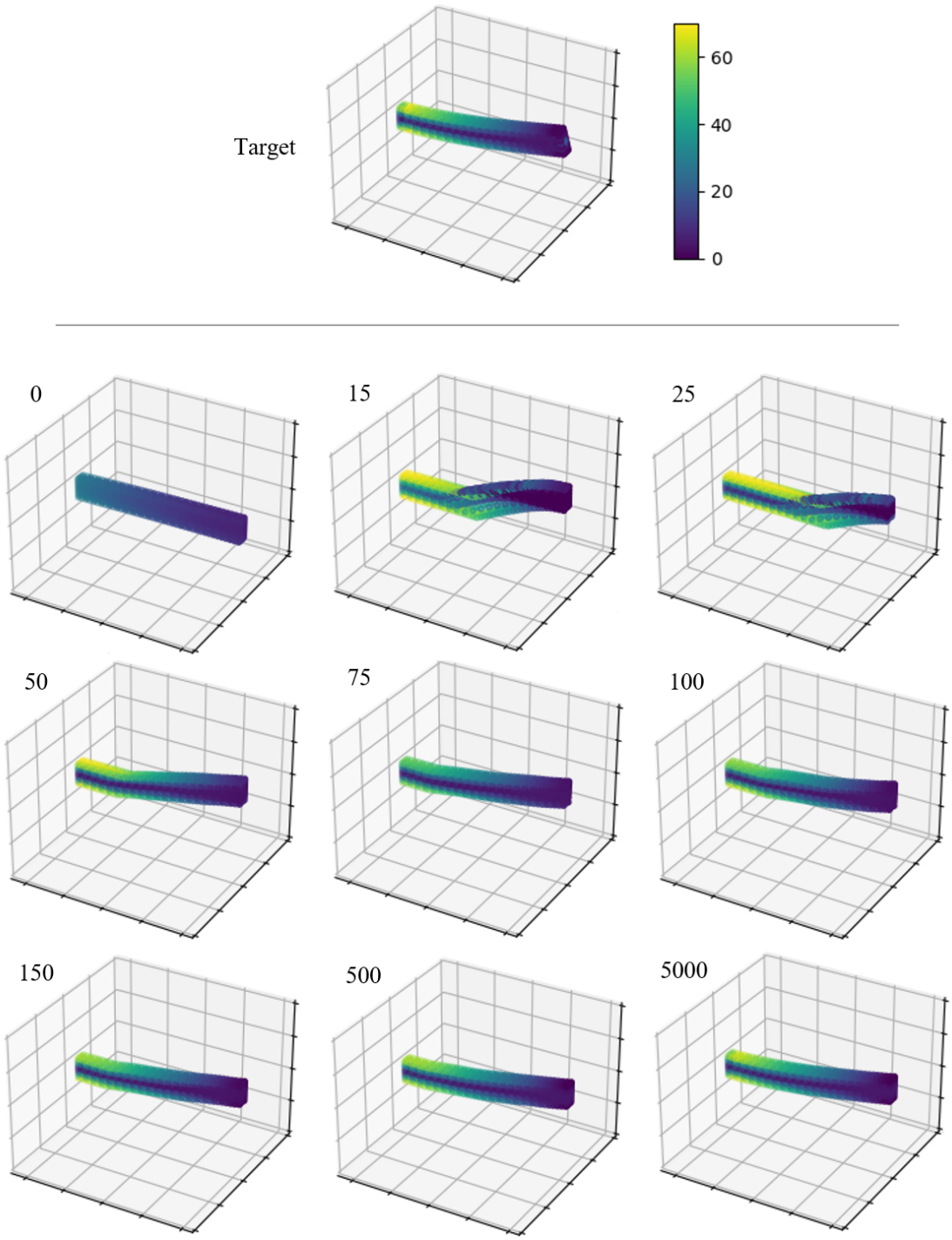


Figure 5.41: Progress of the predicted model for the validation set.

As can be seen in Figure 5.41, the model converges very fast. The loss function has the typical L convergence shape.

5.6.4 Development/Discussion

The architecture of the code has remained mainly the same throughout the development process. From the beginning of development, the biggest change has been to implement scaling of the input features. In the beginning, there was no scaling for neither forces nor coordinates. Furthermore, the forces were inputted in N, and not in kN. Because of the large values, the first predictions for every run were poor, which in turn gave a large loss. Due to the large loss, the calculated gradients were large. The large gradient caused large updates in weights. This effect continued to go on, resulting in an unstable network that was unable to predict anything. This is known as *Exploding Gradients*. To understand why the network was unable to learn, we stripped the code down to be as basic as possible, but it still gave the same predictions. This left the dataset as the problem, and, in the end, a solution was found. When the forces were normalized with the StandardScaler, the results improved greatly. In addition, the input coordinates and vectors were rotated and scaled. Since ML works by learning patterns, it is favorable to have an uniform set of input features. This meant that by rotating and scaling the input features, the model was provided with a solvable problem.

As can be seen in Figure 5.40 the model performs accurately in prediction of the stresses. To also optimize the prediction for displacements, some work on the network or changes in the data has to be implemented.

5.7 Case Study 6: FEA of a Simple Steel Connection with ML

This case study is an extension of Case Study 5. In this case, a simple steel connection has been investigated; see Figure 5.42. It has one arm in each quadrant, with the x- and y-axes shown with a green and red line in the figure. The connection is fixed for one of the arms, while the other three are loaded with a shear force, a normal force, and a moment. Additionally, a vertical force is applied in the middle of the connection. This model tries to represent the response of one connection in a grid of connections. The forces applied at the end of the beams represent the internal forces of the neighboring members. ML has been used to predict the displacements and stresses of this connection, the same as in Case Study 5. To evaluate the full grid of connections, the rotational stiffness of each connection would need to be calculated. This is further investigated in Case Study 7. The connection in this case could be seen as four of the cantilevers investigated in Case Study 5.

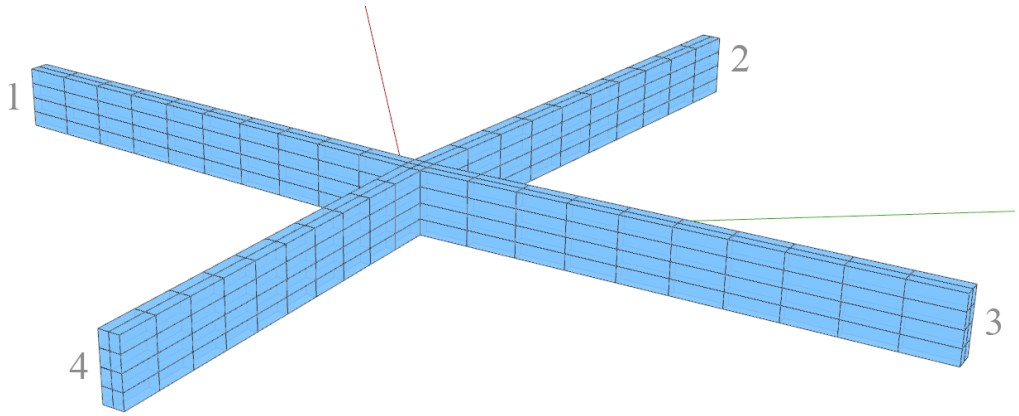


Figure 5.42: Geometry of the simple steel connection used in Case Study 6.

Grasshopper has been used to create the dataset, using the SolidFEM plugin described in Section 4.2. In the datasets created, there are one file for input features, and five files for targets, one for each arm of the connection, in addition to the middle section. As in Case Study 5, the input files consist of vectors that describe the geometry of the beams and the forces applied to each of the free ends. The target files consist of displacements in the x, y and z directions, stresses, and coordinates for the corresponding points. The framework described in section 4.3 has been used to train the model. For this section, the creation of data in Grasshopper will be described first and then the code for machine learning. After this, the final results will be presented before the development that led to these results is discussed.

5.7.1 Dataset creation in Grasshopper

The creation of datasets in Grasshopper can be divided into five parts: Create geometry, mesh geometry, FEA, post-processing, and writing data to file. A flowchart of the Grasshopper algorithm is shown in Figure 5.43. As in Case Study 5, the same custom C# script is used to loop through the grasshopper code.

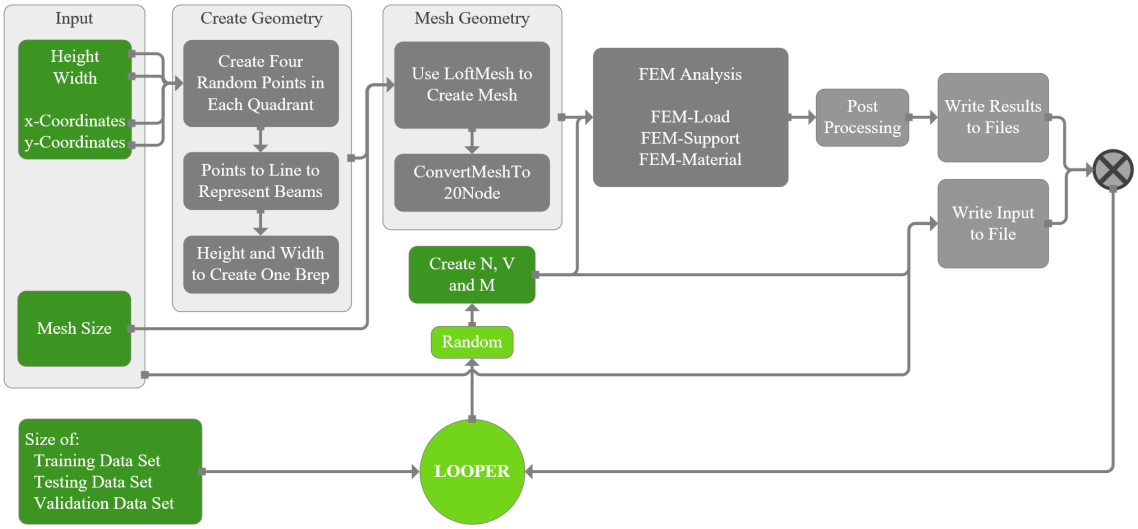


Figure 5.43: Flow chart of the dataset generation in Grasshopper.

Create geometry

To create the geometry, four points are created with random x and y coordinates. The coordinates are created by a random component that picks a number between 1000 and 2000 depending on the seed. Using the iteration of the loop component as the seed, a different case could be created for each loop. In this case, the geometry of the beams is fixed. To create a connection with beams in four different directions, four points are created, one in each quadrant. This is done by multiplying either the x or y coordinate with -1 according to Table 5.25.

Quadrant	Sign for x-coordinate	Sign for y-coordinate
1	+	+
2	-	+
3	-	-
4	+	-

Table 5.25: The sign of x and y coordinates in the four quadrants.

These points were connected to origo with lines to represent the beams. The lines are further expanded with the width and height given in Table 5.26, which also gives the length of the four different beams according to the numbering in Figure 5.42. This results in a Brep that represents the whole geometry of the connection.

Width	Height	L1	L2	L3	L4
100 mm	300 mm	2302 mm	2260 mm	2310 mm	1714 mm

Table 5.26: Model: steel beam connection.

Mesh geometry

From this simple Brep, surfaces are created that describe the four different beams and the middle section. These are used separately in the **LoftMesh** component from the solver described in Sec-

tion 4.1, to mesh the entire geometry. This creates five different lists of 8-node meshes. These are gathered in one long list and sent through the **ConvertMeshTo20Node** component to create a long list of 20-node meshes. The total number of elements is 336. This is a coarse mesh, but since the elements used are quadratic, the accuracy is good. The reason for not using a finer mesh is the time it takes to perform an analysis and, in turn, create the entire dataset. If the analysis takes ten seconds, it would take almost 14 hours to create a dataset with 5000 cases.

FEA

The list of meshes was then used to perform the analysis. Loads and supports were created with the **FEMLoad** and **FEMBoundaryOnPoints** components. One of the beams has a fixed support at the end, whereas the three others have a free end. These free ends are loaded with a normal force, a vertical force, and a moment. In addition, a vertical point load is applied in the middle of the connection. This load is a constant 40 kN in the negative z-direction. The other forces vary and are listed in Table 5.27. To act as the internal forces of neighboring beams, the shear force is applied in the positive z direction, and the normal force is applied as compression. The moment is applied with a force couple, with tension at the top of the beam and compression at the bottom. For the material, the same steel material used in Case Study 5 has been used. The properties of the material are shown in Table 5.28.

Shear force (V)	Normal force (N)	Moment (M)
0 to 50 kN	0 to 50 kN	0 to 7.5 kNm

Table 5.27: Shear force, normal force, and moment applied.

Youngs Modulus	Poisson ratio	Yielding stress	Material weight
210000 MPa	0.3	355 MPa	7850 kg/m^3

Table 5.28: Steel material properties.

Postprocessing of results

Postprocessing of the results consists mainly in sorting the displacements and stresses into the five different parts of the connection. Since the output from the solver is long lists of all the displacements and stresses, it is necessary to sort them into the four beams and the middle section. This is done using the **Point In Brep** component in Grasshopper; this creates a dispatch pattern that can be used for both displacement lists and stress lists. The displacements, stress, and corresponding coordinates are then concatenated into one list for writing to file.

Writing data to file

To create datasets, the same Python script is used to write to files, see Listing 5.5. For this case, the results of the analysis are divided into four beams and the middle section, and the same applies when writing to files. The dataset ends up being five files for the results and one file for the inputs.

5.7.2 Training

The created datasets are then used in Visual Studio Code and PyTorch as input to a ML model. In the following section, the differences from the framework described in Section 4.3 are described. The code is much the same as for Case Study 5.

For this case, the input features are a little different from Case 5 since the geometry is fixed. To represent this case, three different loads are needed, in addition to the coordinates of the resulting vertices and the beam index. From the input file, the forces are loaded and scaled in the same way as in Case Study 5. Since there are five different result files, it is necessary to loop through all of them. For each file, an angle is calculated, this process is similar to the angle calculations in Case Study 5. The angle is then used to rotate the corresponding beam coordinates. The data are sorted in to three lists, stresses, displacements and coordinates. In addition, an index list is created to keep track of the beam index for the corresponding data. This index corresponds to the beam numbering in Figure 5.42, with the middle section having index 0. There is also an additional check for duplicate points, because the middle section shares points with all the other beams. The same scaling used in Case Study 5 is also applied here. The **CustomDataset** component returns the scaled forces, the scaled and rotated coordinates, the original coordinates (for plotting), the target stress, the target displacement and a list of indexes.

The rest of the codes are mostly the same as in Case Study 5. Trainer and network are the same, but with different values used for learning rate, batch size, and layer sizes. This case used **SmoothL1Loss** instead of the normal **L1Loss**. The smooth loss is the same as L1 when the loss is greater than one, but when it is less, the smooth loss is equivalent to **MSELoss**, which is the mean squared error. This is shown in Equation (5.6), with $\beta = 1$.

$$l_n = \begin{cases} 0.5(x_n - y_n)^2/\beta, & \text{if} |x_n - y_n| < \beta \\ |x_n - y_n| - 0.5\beta, & \text{otherwise} \end{cases} \tag{5.6}$$

A sweep run was performed with different layers, learning rates, and dropout values. The optimal configuration from this sweep run is shown in Table 5.29.

Dropout	Layer sizes	Start learning rate	Validation loss
0	[128 512 512 512]	0.024	1.81

Table 5.29: Sweep run from WandB.

5.7.3 Results

The network described in previous sections was used to train a model on the dataset of connections. The results are presented in the following figures. In Table 5.30, the stress, displacement and total validation loss can be seen.

Validation displacement loss	Validation stress loss	Validation loss
1.613	0.1978	1.811

Table 5.30: Resulting validation loss.

Figure 5.44 shows the target and prediction with stresses and displacements. As for the results in Case Study 5, the loss values must be seen in context with the predictions made. In the following results, the displacement loss is higher than the stress loss. From Figure 5.44, it is clear that the predicted stress is better than the predicted displacement, but the difference in loss also comes down to the loss function used. Since the stress loss is less than 1, its loss is squared, creating an even larger difference between the two.

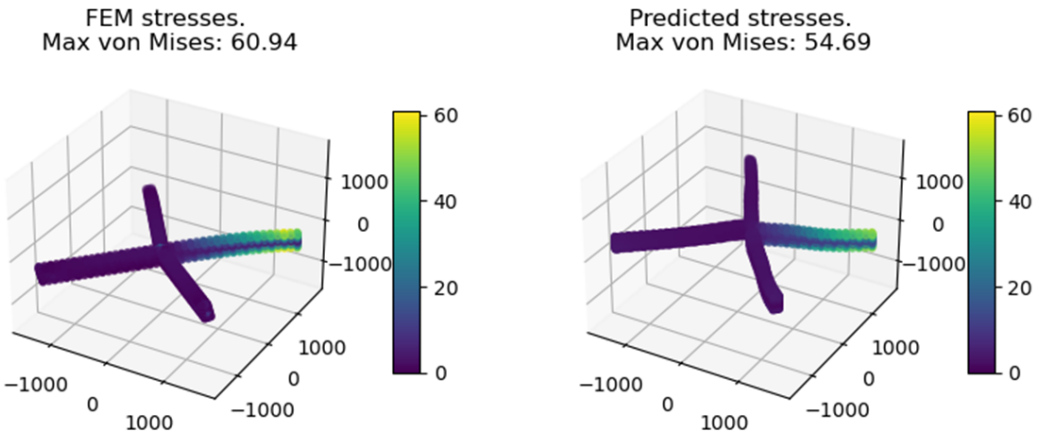


Figure 5.44: Target and predicted displacements and stresses.

5.7.4 Development/Discussion

With the experience of the simple cantilever case, the forces and coordinates were scaled from the beginning of the development process. The architecture of the network was also kept quite similar, since the idea was that this case could be seen as five of the cantilever cases.

In the early stages of development, the coordinates for the beams were not rotated, but were scaled. This meant that when creating an algorithm, the ML tried to predict displacement and stresses, depending on a points x and y coordinates inside an imaginary area, as depicted in Figure 5.45. The problem with this approach was that, towards origo, a small change in coordinates could give a large change in results, again see Figure 5.45. Two solutions to this problem were suggested. Both involved a division of the model into five different parts.

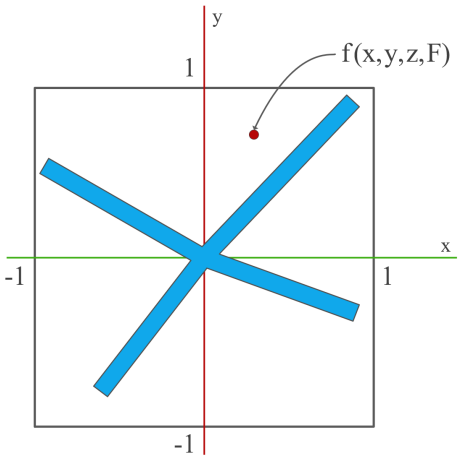


Figure 5.45: Figure of connection before rotation of arms.

The first suggestion was to load the results into five different lists in the **Dataset** component. This meant that instead of six input features: N, V and M forces, and x, y and z coordinates, the network would have 30 features: N, V and M forces, and x, y and z coordinates for five different parts. When trying to implement this, a problem arose. The dimensions of the different arrays did not match since the middle section has fewer vertices than the four beams. This solution was also discussed to be slow since it had an increase in input features. This would require more GPU memory than the other solution.

The other solution was to still have all results in a long list, but add an index to each result. This index refers to what part of the connection to which the result belonged. As described earlier in this section, this is the solution that was chosen. It required only small changes to the algorithm. When this solution was applied, the results improved. In addition to dividing the connection, each part was scaled, and rotated to the x-axis. This meant that the algorithm could predict displacements and stresses for a normalized grid, depending on the forces applied and the index of the beam.

From the result illustrated in Figure 5.45 it is clear that the stress predictions are decent while the network needs some adjustments to get an acceptable prediction of the displacement. This result was not surprising because the codes used here are based on the same code used for Case Study 5 where the results were similar.

5.8 Case Study 7: Rotational Stiffness of Beam-to-Column Connections with ML

The purpose of incorporating rotational stiffness computation is to be able to create a framework for structural analysis of a complete grid of connections. This framework would streamline and reduce the computational cost of the analysis, using an ML model to predict the rotational stiffness for every member of all connections and another model to predict the total behavior of the loaded grid.

In this case study, two different models of steel beam-to-column connections are investigated. The first part of the case study involves a model of a connection with an HEA 300 cross-section. This model is used to verify the rotational stiffness computation with SolidFEM, where the results are compared with Robot. In the second part, a parametric model of beam-to-column connection with solid rectangular cross-section is developed. For this case, with rectangular cross section, datasets are created in Grasshopper and used to train an ML model that predicts the rotational stiffness of the connection.

5.8.1 HEA 300 - Beam to Column Connection

The first case consists of a beam connected to a column, where both parts are HEA 300 profiles of steel type S355. The connection is restrained in all directions on the end surfaces of the column and loaded with a shear force at the free end of the beam. This model is made to further verify the accuracy of the SolidFEM plugin and to evaluate the rotational stiffness calculation of a solid steel connection using SolidFEM. The results are compared with Robot, which has built-in software for steel connection analysis using Eurocode equations. As mentioned in Section 2.3.3, the equations for rotational stiffness calculation in Eurocode are only valid for joints connecting H and I sections. Hence, HEA 300 is the chosen cross-section for the beam and column.

Cross-section	Steel type	Beam length (L_b)	Column length (L_c)	Total shear force (V)
HEA 300	S355	500 mm	966 mm	100 kN

Table 5.31: Setup of the beam-to-column connection.

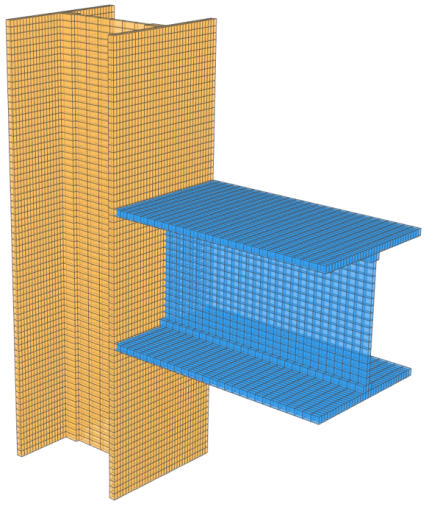


Figure 5.46: The beam-to-column connection in Grasshopper.

Grasshopper

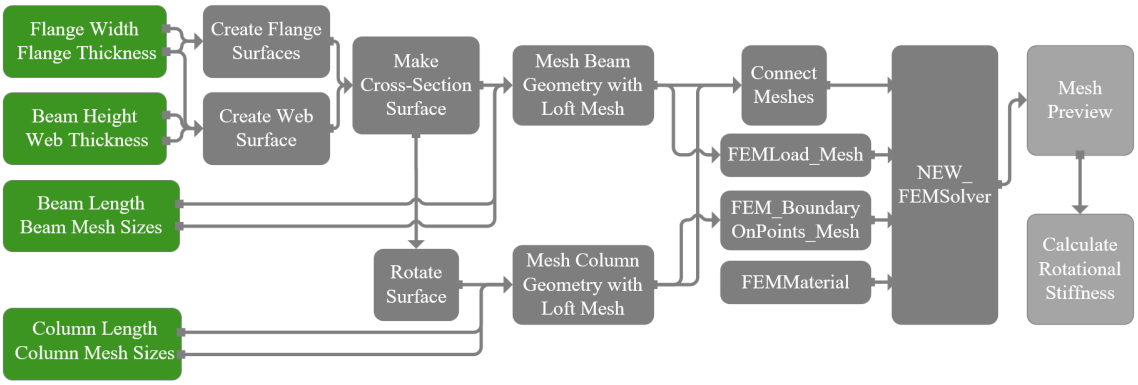


Figure 5.47: Flowchart of the HEA 300 beam-to-column connection in Grasshopper.

The Grasshopper model is an assembly of several parts, a flow chart of the model is shown in Figure 5.47. First, the cross-section of the beam is created as a collection of surfaces. This collection consists of seven surfaces; two surfaces for the flanges, one surface for the web and four small surfaces for the flange-web fillet connection. The flange surfaces are the foundation of the cross-section surface and have the exact dimensions as the standard HEA300 section. To create the web surface, the size of the mesh needs to be considered. This is because the **Loft Mesh** component, which is used to mesh the model, only creates uniform meshes, and the **NewFEMSolver** component has a limitation of number of nodes in its analysis. To be able to analyze this geometry, the 8-node element has been used, in stead of the 20-node, to reduce the number of nodes. Furthermore, this causes the dimensions to deviate slightly from the standard HEA300 cross section. This can be seen in Figure 5.48. For example, the width of the flange (300 mm) is divided into 35 elements so that the web thickness (8.57 mm) becomes as close as possible to 8.5 mm. These dimensions are necessary to make the beam mesh and the column mesh compatible with each other.

These surfaces, which make up the entire cross-sectional surface, are individually duplicated and moved along the longitudinal direction to set the length of the beam. The **Loft Mesh** component takes these surfaces as input, along with the parameters to control the mesh size, to create the compatible beam mesh. To make the column, the cross-sectional surfaces of the beam are duplicated and rotated, and the same procedure is performed for the meshing of the column. All the different 8-node meshes are then merged into one list of meshes. The meshed model can be seen in Figure 5.46.

To mimic a concentrated load at the beam tip, the free beam end is loaded with a total of 100 kN in the z-direction, which is evenly distributed on all nodes of the web. For boundary conditions, all nodes on both end surfaces of the column are restrained in the x, y and z directions. Loads, supports, material properties and mesh list are inserted in the **NEWFEMSolver** component to run the elastic analysis.

To calculate the rotational stiffness, the approach described in Section 2.3.3 is applied. The beam axis and the column axis are constructed from the mesh geometry. An evaluation point is placed on the beam axis a distance of 200 mm from the beam intersection point, and two evaluation points

for the column are placed on the column axis a distance of 140 mm from the columns intersection point. See Figure 2.10 for an additional explanation on the placement of the points.

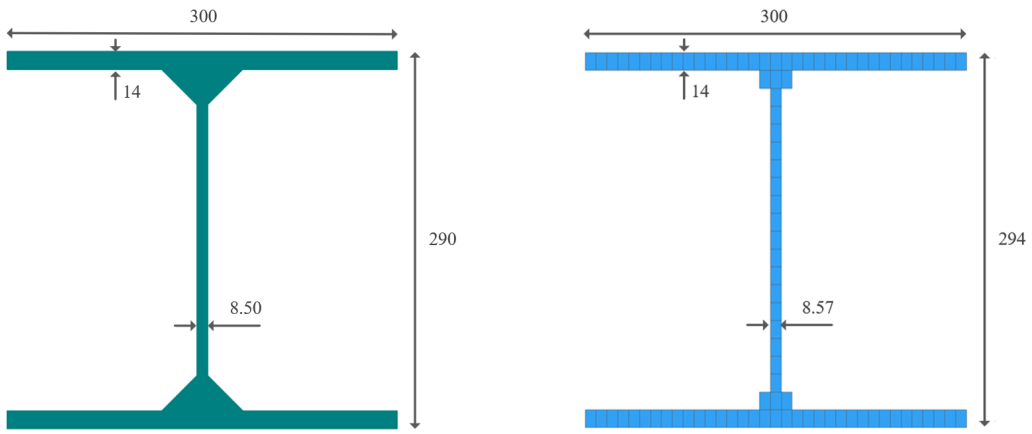
Abaqus

The connection is made as two separate parts in Abaqus. The exact same dimensions used in Grasshopper is also used in Abaqus. To mesh the two parts, the edges are given a seed to achieve the same amount of elements. The 8-node hexahedron elements (C3D8) are chosen, which is equivalent to the type of elements used in the SolidFEM analysis. The two parts are connected as an assembly by applying face-to-face constraint. For boundary conditions, the end surfaces of the column were restrained. The free-end web nodes of the beam were applied with concentrated force in the negative z-direction.

Robot

The structure in Robot is drawn as lines. The column and beam are assigned with HEA 300 cross-sections from Robots section database, and the material is set to steel type S355. Boundary conditions are applied to the end nodes of the column line, where the two nodes are set as fixed. Two load classes are created; one for dead load (self-weight) and one for live load. A concentrated force in the z direction, with the given magnitude, is placed on the node on the free end of the beam. A manual combination of the two load classes is generated, where both classes are applied with a partial safety factor of 1.0, to make the load situation similar to the Grasshopper model.

To calculate the rotational stiffness in Robot, their built-in steel connection design is used. A new steel connection is defined and the beam-to-column type is selected. The connection is applied with welds on both the flanges and the web. After the automatic calculations are completed, the rotational stiffness $S_{j,ini}$ could be retrieved from the results report. In these calculations, Robot uses the Eurocode equations, which are described in Section 2.3.3.



(a) HEA300 dimensions in Robot. **(b)** Cross-section dimensions in Grasshopper

Figure 5.48: Dimensions of the cross-sections

Results

Comparison with Abaqus: Displacement and von Mises stress

To compare the displacements and von Mises stresses of the connection from SolidFEM and Abaqus, a node on the free end of the beam is selected. The values provided in Tables 5.32 and 5.33 are all extracted from the node highlighted in yellow in Figure 5.49 and Figure 5.50.

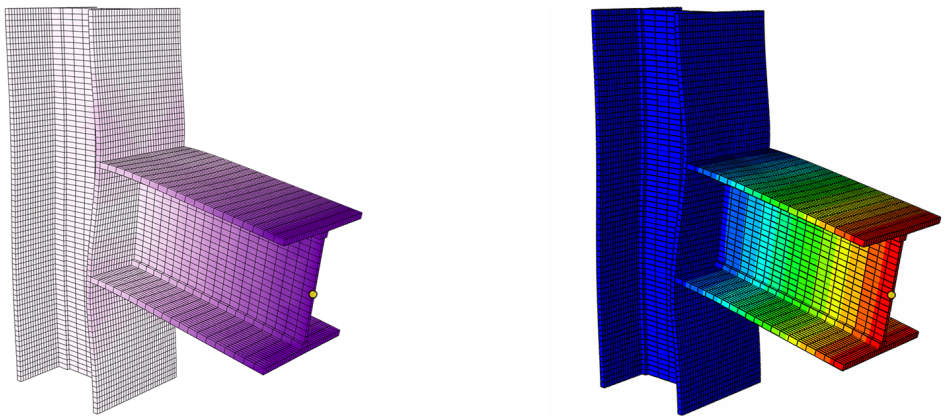


Figure 5.49: Comparison of displacement: SolidFEM (left) and Abaqus (right).

Displacement in SolidFEM	Displacement in Abaqus	Difference
-0.920 mm	-1.035 mm	11.11 %

Table 5.32: Displacement in z-direction: SolidFEM and Abaqus.

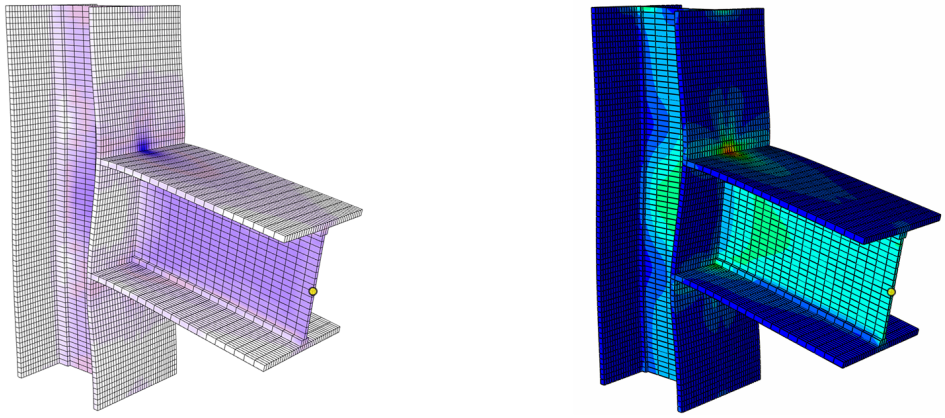


Figure 5.50: Comparison of von Mises stress: SolidFEM (left) and Abaqus (right).

Von Mises stress in SolidFEM	Von Mises stress in Abaqus	Difference
64.976 mm	65.300 mm	0.50 %

Table 5.33: Von Mises stress: SolidFEM and Abaqus.

Comparison with Robot: Rotational stiffness

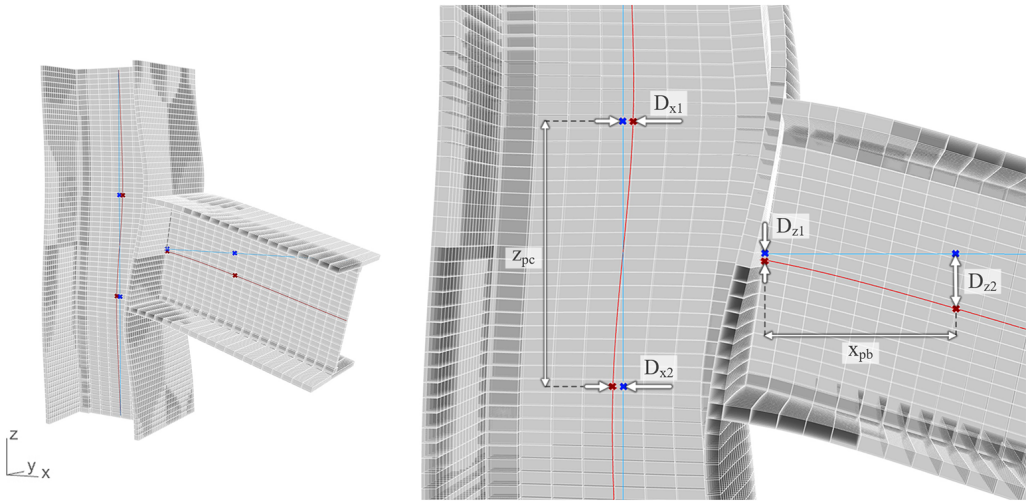


Figure 5.51: Computational model for rotational stiffness with SolidFEM in Grasshopper. The blue lines are the original axes of the beam and the column, and the red curves are the axes after deformation.

From the points in Figure 5.51, the distances are measured to obtain the values needed for the calculation of rotational stiffness.

The angle of rotation of the beam:

$$\phi_b = arctan\left(\frac{D_{z2} - D_{z1}}{x_{pb}}\right) = arctan\left(\frac{0.354252\,mm - 0.038559\,mm}{200\,mm}\right) = 0.001578\,rad \quad (5.7)$$

The angle of rotation of the column:

$$\phi_c = arctan\left(\frac{D_{x2} - D_{x1}}{z_{pc}}\right) = arctan\left(\frac{0.066856\,mm - 0.066851\,mm}{280\,mm}\right) = 3.5714 \cdot 10^{-8}\,rad \quad (5.8)$$

The initial rotational stiffness of the connection analyzed with SolidFEM is thus:

$$S_{j,ini} = \frac{M_j}{\phi_j} = \frac{(V \cdot L_b)}{(\phi_b - \phi_c)} = \frac{100\,kN \cdot 0.5\,m}{0.001578\,rad - 3.5714 \cdot 10^{-8}\,rad} = 31\,685.68\,kNm/rad \quad (5.9)$$

The automatic calculations of the initial rotational stiffness from Robot:

$$S_{j,ini} = \frac{Ez^2}{\sum_i \frac{1}{k_i}} = \frac{210\,000\,N/mm^2 \cdot (276\,mm)^2}{\left(\frac{1}{5.129\,mm} + \frac{1}{6.258\,mm} + \frac{1}{6.258\,mm}\right)} = 31\,088.58\,kNm/rad \quad (5.10)$$

SolidFEM		Robot		Difference
31 685.68	kNm/rad	31 088.58	kNm/rad	1.92 %

Table 5.34: Comparison of initial rotational stiffness: SolidFEM and Robot.

Discussion

Compared to Abaqus, the von Mises stress values are similar, but the displacement differs by 11.11% in the yellow node. This may be due to the way the two parts were connected in Abaqus. In SolidFem, the connection acts as a complete part, unlike in Abaqus, where the beam and the column were connected as an assembly by applying a face-to-face constraint. When connecting them with constraints, there will be a duplication of nodes. This means that the model analyzed with SolidFEM and the model in Abaqus will not have exactly the same number of nodes. Although they have the same number of elements, the difference in number of nodes will affect the calculations. This may have influenced the results. In Figure 5.50, it can be seen that there is a slightly finer transition in the stresses between the column and the beam for the SolidFEM model than for the Abaqus model.

The rotational stiffness of the connection calculated using Grasshopper and SolidFEM is in line with the calculation performed in Robot. From the calculation in Equation 5.8 it can be seen that the angle of rotation of the column has little impact on the rotational stiffness. When determining the angle of rotation of the beam, the evaluation length, x_{pb}, should be considered. The evaluation point should be placed at the beginning of the region where the displacement curve of the beam is close to linear. **There is no documentation on exactly where this point should be placed**. In this case, from around 180 mm and to the end of the beam, the curve is almost linear. A length of 200 mm was chosen to ensure that the point was in the linear part.

As can be seen in Figure 5.52, the values for the rotational stiffness with SolidFem are close to the Robot results when the evaluation point is placed in the early linear area. If the evaluation point was placed with x_{pb} = 223 mm, the calculation of rotational stiffness with SolidFEM would give the exact same value as Robot. The Eurocode equations, which Robot uses, are generally conservative, it was therefore expected that the numerical method initially gave a slightly higher value.

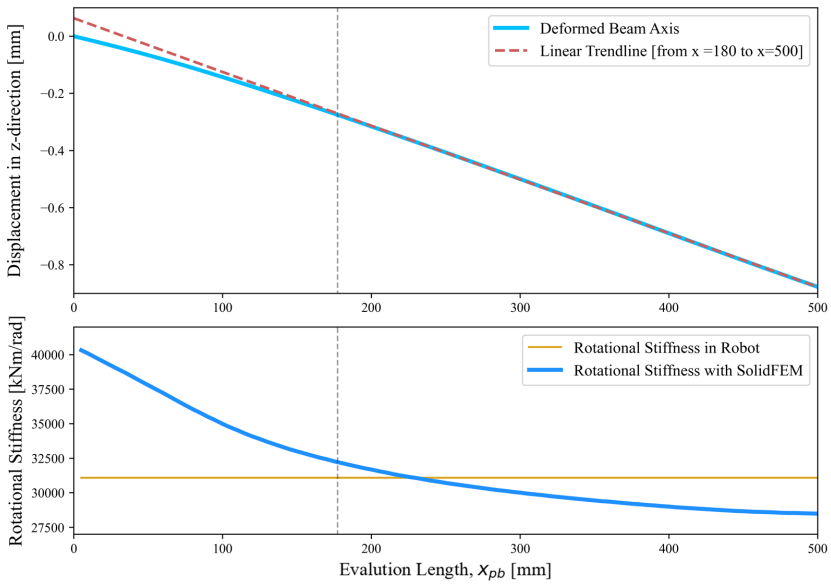


Figure 5.52: Beam deflection curve and the rotational stiffness at different evaluation lengths.

5.8.2 Rectangular Beam-to-Column Connection with ML

In this part of the case study, a Grasshopper model of a beam-to-column connection with rectangular cross-sections is created. With this Grasshopper model, datasets are created to train an ML model. The ML model is trained to predict rotational stiffness of connections, when the beam height and the force magnitude varies. The rotational stiffness is calculated with the same method used in the first part of this case study. The connection is restrained by the end surfaces of the column, and a shear force is applied and distributed at the free end of the beam. This can be seen in Figure 5.53, where the yellow arrows indicates the distributed shear force and the small gray spheres represent the fixed nodes. The dimensions and the magnitude of the shear force are listed in Table 5.35 and 5.35.

Steel type	Beam Length	Column Length	Total shear force
S355	500 mm	1200 mm	10 to 50 kN

Table 5.35: Setup of the beam-to-column connection with rectangular cross-sections.

Beam width	Beam height	Column Width	Column height
100 mm	100 to 200 mm	100 mm	300 mm

Table 5.36: Dimensions of the cross-sections of the beam and the column.

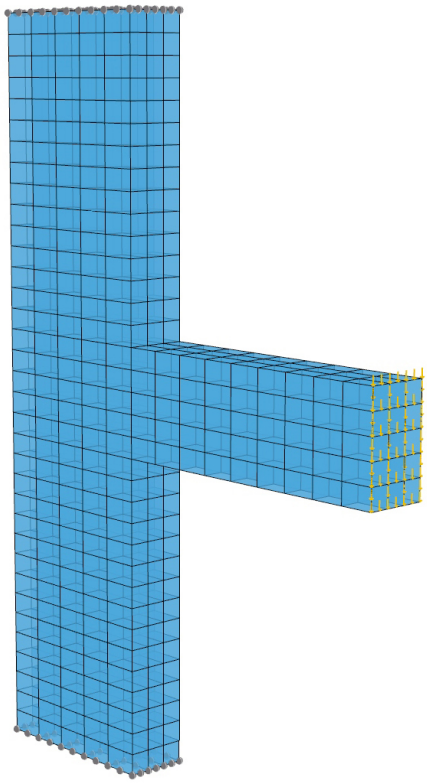


Figure 5.53: Grasshopper model of the beam-to-column connection with rectangular cross-section.

Dataset creation in Grasshopper

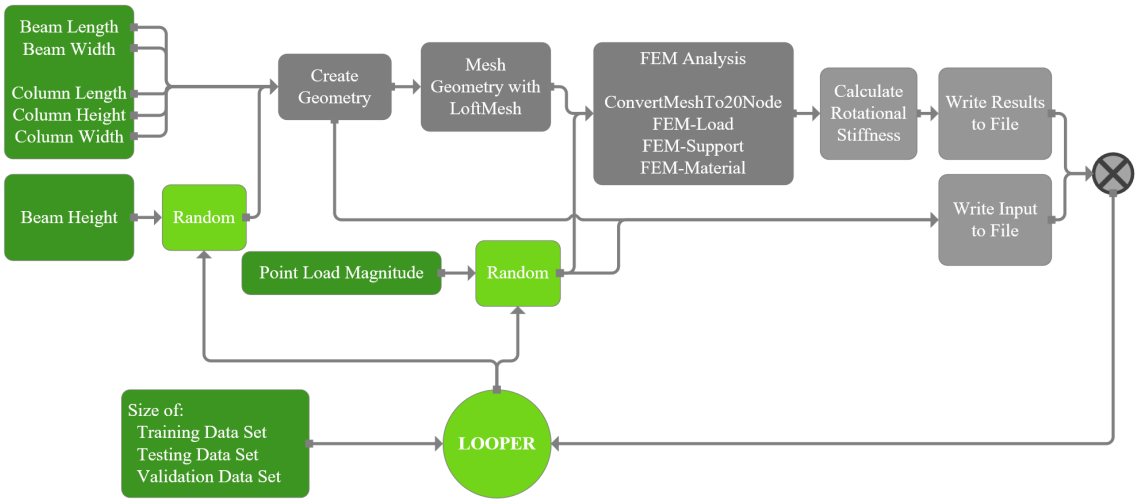


Figure 5.54: Flowchart of dataset generation in Grasshopper.

The process of generating the dataset is shown as a flowchart in Figure 5.54. The geometry of the connection is created with the same approach as for the connection with the HEA300 cross-section, described in Section 5.8.1. The cross-sections of the beam and the column are first created as surfaces. The cross-section of the column has fixed dimensions, but the height in the beam cross-section varies randomly. This random component picks a number between 100 and 200 mm, depending on the seed given from the loop component. This custom loop component is explained in Section 5.6.1. The cross-section surfaces are duplicated and moved along their longitudinal direction to set the length. With these surfaces, the **Loft Mesh** component is used to mesh the geometry, and the **ConvertMeshTo20Node** component converted the beam and column meshes to 20-node meshes.

Nodes for support and load are extracted from the meshed geometry. The shear force is evenly distributed and applied to all nodes on the free end beam surface. The magnitude of the force is connected to a random component that ranges from 10 to 50 kN. The meshes, the load, the support and the steel material are inserted in the **NEWFEMSolver** component, and the FEA is performed.

For creating the datasets, the loop component along with the Python script that writes data to a text files are used. The Python script is illustrated in Listing 5.5. For each loop, the model is given a new value for the beam height and for the magnitude of the shear force. In this way the new geometry for the beam is automatically constructed and the analysis is performed. The rotational stiffness of the connection is thereafter calculated. At the end of each loop, the new beam height, the new force magnitude and the corresponding rotational stiffness are written to a text file. The beam height and the magnitude of the shear force are used as input to the ML model, while the rotational stiffness is used as target for the ML model. In this code 4000 datasets are created for training, while the validation and testing has 500 samples each.

ML model

The created datasets are used to train an ML model with the task of predicting the rotational stiffness of connections. To create the ML algorithm, the same framework used for the other ML models in this thesis is applied. This method is described in Section 4.3. The network architecture used for this problem is a fully connected network. The input layer consist of a vector with two input features, two scalars that represents the force magnitude and the beam height. The target in this case is a single scalar representing the rotational stiffness of the connection.

For this problem, to avoid exploding gradients, the input features are scaled with the help of the MinMaxScaler provided by the *scikit-learn* package. This scaler normalizes the data between zero and one. Additionally, the rotational stiffness is scaled to improve predictions. To illustrate the function that the network is going to solve to predict the rotational stiffness, Figure 5.55 is included.

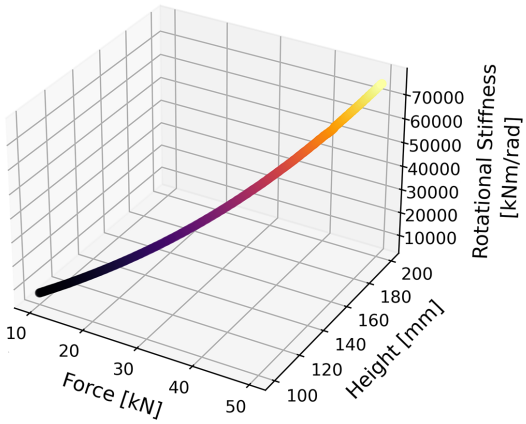


Figure 5.55: The datasets for rotational stiffness visualized as a 3d graph.

For this problem, not only the losses are logged to WandB, the actual predictions of the rotational stiffness are logged together with the target. This can be seen in Figure 5.56, where the green graph represents the prediction and the orange line is the target.

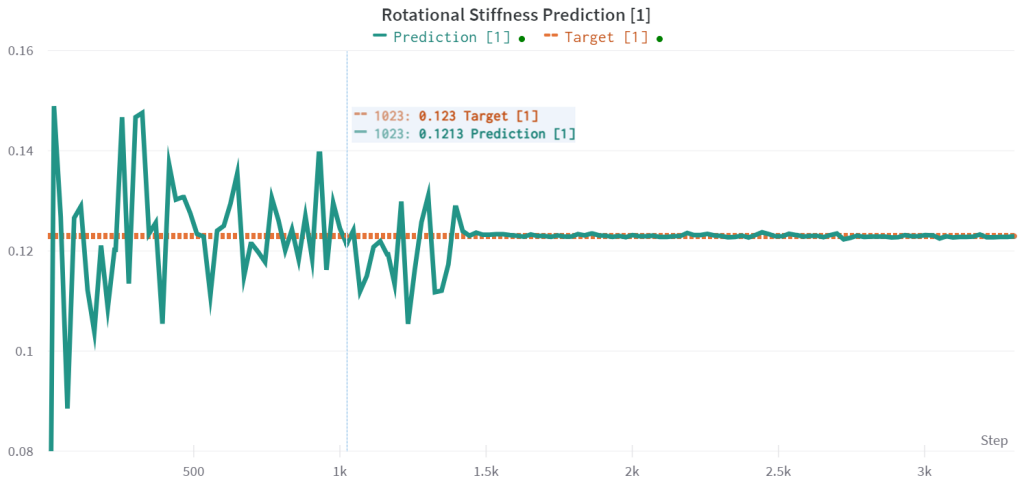


Figure 5.56: Plot of rotational stiffness prediction and its target throughout a run.

Results

A sweep run is completed to find the optimal configurations. The result from the sweep is presented in Table 5.37.

Dropout	Layer sizes	Start learning rate	Validation loss
0	[64 128 64 128]	0.009	0.00036

Table 5.37: Sweep run from WandB.

By prolonging the run for the best configuration, the validation loss listed in Table 5.38 is achieved.

Validation loss
0.000269

Table 5.38: Resulting validation loss.

Discussion

Shown in Figure 5.55, the relationship between the rotational stiffness and the input features produces a smooth 3D curve. It should therefore be a manageable task to train the ML model to produce an accurate prediction of the rotational stiffness. In our first models, the network was unable to make accurate predictions and the losses were dispersed. Due to our experience from Case Study 5, the input features and the target for this network are scaled. After scaling the data, the predictions of the network improved.

The validation loss, from Table 5.38, represents the mean difference between the scaled prediction and the scaled target. With the correct unit for rotational stiffness, this equates to 26.9 kNm/rad. The calculated rotational stiffness for the different geometry ranges from approximately 10 000 kN to 70 000 kN. This means that an average prediction misses with 26.9 kNm/rad, which is an acceptable estimate. For the lowest calculated rotational stiffness, this corresponds to a estimation error of 0.25 %, a small error in these kind of approximation tasks.

6 Discussion/Conclusion

6.1 Discussion

This section will answer the research questions in light of the different case studies that have been explored. The section is divided into two main parts: one part concerns the implementation of FEA with higher-order volumetric finite elements in Grasshopper and the other part concerns the application of ML on the different aspects of the FEA. The benefits of the developed software will be discussed, in addition to looking at the problems that were encountered and how to further develop and improve the software.

6.1.1 Solid FEM

Efficiency is key in the development of designs as a structural engineer. The use of digital software has made this easier over the last few decades and the use of AAD even more so in the last couple of years. For a design to be good, it must be both aesthetically pleasing and structurally sound. The introduction of FEA in an AAD environment creates better opportunities for the latter. By exploring the research question the benefits of implementing volumetric finite elements in AAD are highlighted.

The investigation began with a simple FEM solver that worked only with eight node elements. This solver proved the possibilities of implementation and illustrates that the benefits are huge. Compared to Abaqus, where it is a tedious process to change the geometry of a model, with an FEM solver in Grasshopper, changes can be made instantaneously and the result is updated in real time.

Furthermore, the expansion to elements with higher-order shape functions, with the development of the SolidFEM plugin, illustrates the same positive effects. The accuracy of the solver was improved, as well as the inclusion of different load and support types. This made the solver applicable to more cases. Case Study 2, with the arched beam, showcased the benefits of elements with higher-order shape functions when the geometry became more complex and, in this case, curved. This case study indicated the possibilities of the SolidFEM plugin. For designs with complex geometry and especially intricate connections, the solver opens the possibilities of optimizing the geometry much faster than standard FEM software. This is especially relevant for conceptual structural design. In more traditional structural design, the structural elements are often standard and easily created in dedicated FEM software. Designs that have meaning beyond being structurally sound often have more complex geometry, and the analysis of these structures would benefit from elements of higher order.

The issues that were encountered when working on Case Study 2 were mainly connected to the preprocessing, the meshing. As geometry became more complex, meshing became harder and more computationally costly. To further improve this solver, a better method to mesh more complex geometry has to be developed. This could be replaced by ML, which will be discussed in the next section.

For these cases, the solver itself had no real issues, but there could always be improvement. There was an issue of exceeding the memory cap for Rhino when creating the stiffness matrix. The way the plugin was built, the whole stiffness matrix was in one continuous array. This meant that the entire array was stored in the short-term memory of the computer, which is a limited resource. With about 1600 elements, there are about 31 500 DOFs, which makes the stiffness matrix array almost 8 Gb. There are different methods for solving this problem and it would be smart to investigate this. This limitation could be a problem for more complex geometries and designs. Another improvement would be the implementation of consistent load vector for surface loads. This was investigated but not fully implemented. The method of load lumping is not recommended for elements with higher-order shape functions. It would give higher accuracy and better results with the method of consistent load vector on surface loads.

6.1.2 Machine Learning

An important part of the ML that applies for the further development of all cases explored would be the integration with Grasshopper. When a model has finished training, it should be possible to create a Grasshopper plugin that uses this model to make predictions in real time. With the input as the input features assigned in the machine learning algorithm, and the output the prediction from the model.

Meshing

Independent of which case study we are talking about, a neural network that tries to predict the target with a continuous function needs data where the target is continuous with respect to the input. The importance of getting familiar with our dataset was one of the first lessons we had to learn the hard way. The need to have a continuous feature-target relationship also reveals one of the largest challenges when it comes to meshing with machine learning. As long as we are able to create a mesh that is continuous with respect to the input object, the methods developed in this thesis are applicable to any object shape. For example, a model could be trained to predict a good mesh of a connection as long as we are able to create data for training where the input is continuous with respect to the target. For a more complex geometry, more points are needed to describe the boundary of the geometry, but the methods developed are the same. That said, there may be different approaches worth exploring, for example, representing the object with a nurbscurve, where the input features are the control points and its weights. Labels could also be represented in a different way. For example, a network could try to predict the coordinates of the mesh vertices directly or use image segmentation to classify each point in the grid into one of the two classes, mesh vertex or not mesh vertex. For now we are going to focus on the methods used in this thesis. As a summary table, 6.1 presents the losses that we managed to achieve with the different networks.

Case study	Dim	Loss
2D fully connected	256	0.01410
2D unet	256	0.00093
3D fully connected	40	0.03265
3D conv + fully connected	40	0.02228
3D conv + fully connected	80	0.02225
3D unet	64	0.02511

Table 6.1: Summary of Validation Losses

From Case Study 3, two of the most promising networks are included in this thesis. Based on the results, it is clear that the Unet architecture is superior to the fully connected architecture. The Unet performs very accurately in predicting the DF. The method developed are close to optimal and the performance is well inside the accuracy needed to predict the location of the mesh vertices.

In the 3D Case Study, due to memory capacity on the GPU we were not able to use the same grid dimensions as for the 2D case. As a consequence, the accuracy of the location of the vertices is not at the same level. One solution to this problem that is not explored in our thesis is to interpolate between all the values in a region around all the local minima. This will open the possibilities to find mesh locations in between the grid points and lead to more accurate predictions of the vertices without needing to increase the number of grid points. This approach is something we would like to recommend for further work. Alternatively to this solution, it is possible to increase the number of points in the grid. As shown in the grid dimension study, this has a positive effect on predicting the correct location of the mesh vertices. An increase in grid points will also improve the DF predictions. It will especially improve the networks that struggle to predict the mid-side nodes due to different number of nodes in each direction.

If fine-tuning the grid density and network depth-with relationship, we are confident that the network architecture with an encoder and fully connected neural network would yield satisfying results. One solution that will make it possible to increase the number of grid points is to use **ResBottleneckBlock** in the encoder instead of the **DoubleConv** class. This will reduce the GPU memory allocated and open up the possibility to increase the grid point density, as we did for the Unet architecture in 3D. Alternatively, the density of grid points could be increased if a GPU with more memory was available.

Another recommendation for further work, and something that would be very interesting to explore, is the further development of the 3D Unet architecture. The process of developing this architecture is started in this thesis, but due to available time, we were unable to make this network perform at an acceptable level. It might be that this architecture is not suited for 3D DF prediction, but based on the superior results from Unet in Case Study 3, it is natural to believe that the Unet architecture could result in a high-performing network also in three dimensions. Something that would be interesting to try out in this case is to, somehow, increase the number of points in the grid.

As described in the Introduction of this thesis, we have focused on exploring and developing different methods for predicting the location of the mesh vertices to find the most optimal method for this task. The methods developed are only the start of creating a machine learning meshing component in an AAD environment. The next step in the development is to create a network that

predicts the connectivity between vertices. Inspiration for this work can be found in Alexis Papagiannopoulos and Avellan, 2020. In this paper, a triangulation algorithm is used to predict the connections. The paper uses a connection table consisting of the probability that two vertices will be connected as targets in a neural network. A similar method could be developed for hexahedral elements. Alternatively, a triangulation algorithm can be performed, before transforming the tetrahedral mesh to a hexahedral mesh as described in Xifeng Gao and Panozzo, 2017. Lastly a component has to be created in grasshopper, this component should be created in a similar way to the FEM solver in this thesis. The component needs to include a code that scales the input geometry to fit the data used for training, then the trained neural network is called from inside the code, and the prediction for the model is scaled back to its original size. The scaling code is similar to the code used to scale the data created for training.

If we had more time to work on our meshing networks, we would also like to include the number of elements as an input feature. For now we have fixed the target number of elements. Note that the number of elements is not the target number in every case, this means that the number of vertices in each direction varies in the data used in our models. A fixed number of elements or vertices in each direction would be a lot easier to predict. Our models are able to predict different numbers of vertices in each direction, which is a property that we want to have in the models. By including the number of elements in the network and using many different values we could have a slider in the final Grasshopper component that decides the density of the final mesh. If the slider was close to zero the mesh would be coarse, or dense if the slider var set to one. Including this variable of elements would be fairly simple and a realistic change to our networks without having to change the network architecture a lot.

The methods developed for predicting the location of mesh vertices clearly perform well inside the accuracy needed to create a regularly shaped element with high performance. For further development, it is clear that Unet should be used in 2D, while the encoder + fully connected network is the best option in three dimensions. The method developed is also transferable to more complex geometry as long as it is possible to create adequate meshes for training the model. In the case of more complex geometry, we need more points to describe the boundary of the object, but the method is the same.

Finite Element Analysis

It is clear from the results in Section 5.7.3 that the current stress predictions are within an acceptable range of the results from the FEA. The displacement predictions deviate to much from the FEA results, improving this would be a priority in future development. This could be done by either improving the network, or redefine the data that defines the problem. Improving an algorithm like this takes a lot of time, since it is mostly a case of trial and error. Different network architectures could be applied, or the features could be sent in to the network in a different way.

The case investigated in Section 5.7 is a simplified version with deterministic geometry. As mentioned in Section 5.7.1, the algorithm for creating data already has the ability to create datasets with randomized geometry. In addition, the height and width of the beams could be randomized. This would make the ML model applicable to a larger range of geometries.

In addition, in its current state, the ML model has all vertices of the meshed geometry as input features. This means that to make a prediction on a new case, it has to be meshed before it can be sent into the ML model. Since all beams are rotated and scaled to the same scale, the input coordinates could be replaced by a standardized grid that represents these coordinates rotated to the x-axis and scaled. With input features that represent the geometry of all the beams, this standardized grid could be rotated and scaled up to correspond to the meshed geometry. By implementing this, the only input features needed would be vectors, height and width, to represent the beams, and the forces applied.

6.1.3 Rotational stiffness

The calculation of rotational stiffness in Grasshopper creates the opportunity to analyze a whole grid, as the one shown in Figure 6.1. Analyzing such a grid of connections, would further showcase the benefits of FEA with higher-order solid elements. With the current FEM software available in Grasshopper, such an analysis would prove difficult, and it is this gap that we would fill by introducing FEA with higher-order solid elements. The rotational stiffness of each connection in the grid would be different, since the stiffness depends on the rotation of the beams and the moment; see Equation 2.38. This means that to calculate the entire grid, a quick method of calculating the stiffness is needed.

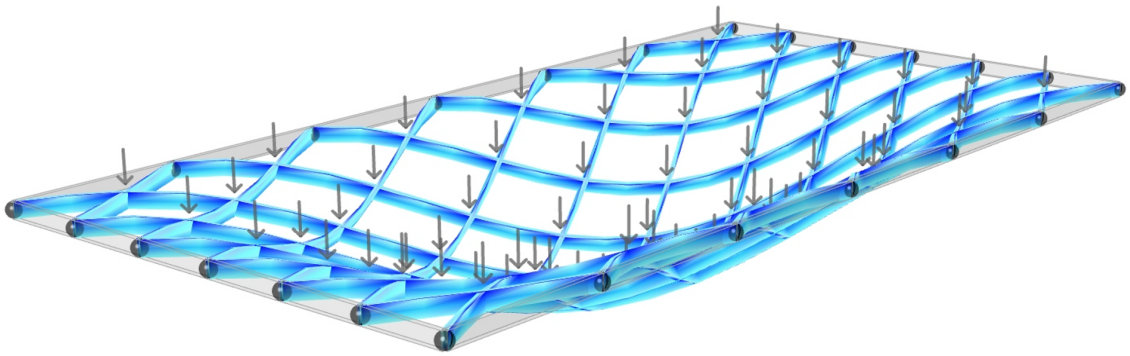


Figure 6.1: An example grid created in Grasshopper

From the calculations in Section 5.8.1, the method applied in Grasshopper is close enough to the Eurocode approach in Robot. Furthermore, the ML approach shows positive results. This means that rotational stiffness could be implemented in the whole ML model to predict stress and displacement. By either first predicting the stiffness and then predicting stress and displacement, or making a full model for everything. With this, a full grid could be analyzed with an ML approach, which would be a lot quicker than a more traditional method.

6.2 Conclusion

What are the possibilities and issues of Finite Element Analysis with solid elements in an Algorithms Aided Design environment?

Can parts of, or the whole, Finite Element Analysis be replaced by machine learning?

The first two case studies illustrate the usage of the FEM plugins developed for Grasshopper and the benefits of implementing the FEA in an AAD environment. One of the greatest benefits of using an AAD environment is to make the design process more dynamic with the possibility to make rapid changes to the design. With the FEM plugins developed in this thesis, we are increasing the value of the AAD process by adding an important element, real-time FEA with solid elements. By implementing elements with higher-order shape function in the FEM solver, we are able to achieve better accuracy of the stress field for designs with complex geometry and connections, especially in load cases with bending. This can be seen in the case studies, where the FEM solver performs accurately in comparison to Abaqus.

One of the most challenging parts of the FEA in an AAD environment is to create an adequate mesh for complex geometry. To solve this issue, we have investigated the possibility of applying ML to this part of the analysis. In this thesis, we have solved one of the most difficult aspects of the meshing process, predicting the location of mesh vertices of an object. The method developed for predicting mesh vertices can be transferred to a more complex geometry than presented in this thesis. There is still some work to be done to get a complete meshing algorithm, but we are confident when we say that the meshing part of the FEA can be replaced by ML.

To highlight the benefits of our work: It would be faster to perform an FEA on a conceptual model created in Grasshopper with our FEM solver than to analyze the same model in Abaqus, by either exporting the model from Grasshopper or recreating the geometry in Abaqus. That being said, our FEM solver is not as fast as the Abaqus solver on the actual analysis. To improve the calculation time of the solver in Grasshopper, it would need to be optimized further. Another solution to this issue is to create an ML model that performs the entire FEA. This task is investigated in Case Study 5 and 6. The results are promising; especially the stress predictions are close to the results from the FEA.

Additionally, the inclusion of rotational stiffness calculations, both in Grasshopper and predicted with ML, shows the possibilities of analyzing more complex structural systems, like the grid in Figure 6.1, in an AAD environment.

Implementation of a FEM solver with solid elements in an AAD environment improves the process of creating conceptual designs, that are both aesthetically pleasing and structurally sound. Furthermore, it is clear that ML can improve this analysis, by improving the mesh generation for complex geometry and reducing the computational time.

Bibliography

Alexis Papagiannopoulos, P. C. & Avellan, F. (2020). *How to teach neural networks to mesh: Applications on 2-d simplicial contours*. Retrieved 22nd May 2022, from https://www.journals.elxevier.com/neural-networks

Autodesk. (n.d.). *Robot structural analysis*. Retrieved 25th May 2022, from https://www.autodesk.no/products/robot-structural-analysis/overview?term=1-YEAR&tab=subscription

Bathe, K.-J. (2014). *Finite element procedures*. Klaus-Jürgen Bathe.

Bell, K. (2013). *An engineering approach to finite element analysis of linear structural mechanics problems*. Fagbokforlaget.

Bharath Ramsundar, R. B. Z. (2018). *Tensorflow for deep learning*. O'Reilly Media, Inc.

Biewald, L. (2020). Experiment tracking with weights and biases [Software available from wandb.com]. https://www.wandb.com/

Brenner, S. C. (1960). *Mathematics of computation*. American Mathematical Society.

Company, S. T. (n.d.). *Abaqus'*. Retrieved 25th May 2022, from https://www.simuleon.com/simulia-abaqus/

Csparse. (n.d.). Retrieved 4th June 2022, from https://github.com/ibayer/CSparse

ECCS, E. C. f. C. S. (2016). *Design of connections in steel and composite structures – eurocode 3 – design of steel structures. part 1–8 design of joints*. Ernst & Sohn.

Eikeland, P. T. (2001). Teoretisk analyse av byggeprosesser. http://v1.prosjektnorge.no/files/pages/362/samspillet-i-byggeprosessen-eikeland.pdf

Eurocode 3: Design of steel structures - part 1-8: Design of joints. (2009). British Standards Institution.

Goodfellow, I., Bengio, Y. & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. https://doi.org/10.1109/MCSE.2007.55

Joints in steel construction moment connections. (1995). The Steel Construction Institute.

Kaiming He, S. R., Xiangyu Zhang & Sun, J. (2015). *Deep residual learning for image recognition*. Retrieved 3rd June 2022, from https://arxiv.org/pdf/1512.03385.pdf

Math.net numerics. (n.d.). Retrieved 4th June 2022, from https://numerics.mathdotnet.com/

McNeel, R. (2022). *Rhino 3d*. Retrieved 25th May 2022, from https://www.rhino3d.com/

Microsoft. (2022a). *Visual studio 2022*. Retrieved 25th May 2022, from https://visualstudio.microsoft.com/vs/

Microsoft. (2022b). *Visual studio code*. Retrieved 25th May 2022, from https://code.visualstudio.com/

Mitchell, T. M. (1997). *Machine learning*. MacGraw-Hill.

Prakhar. (2020). *Intuition of adam optimizer*. Retrieved 20th May 2022, from https://www.geeksforgeeks.org/intuition-of-adam-optimizer/

Preisinger, C. (2013). Linking structure and parametric geometry. *Architectural Design*, *83*(2), 110–113. https://doi.org/10.1002/ad.1564.

Schlömer, N. (2020). *Pygmsh'*. Retrieved 22nd May 2022, from https://pygmsh.readthedocs.io/en/latest/index.htmls

Stanford. (2022). *Convolutional neural networks (cnns / convnets)* [online course, stanford University]. https://cs231n.github.io/convolutional-networks/

Sullivan, B. & Kaszynski, A. (2019). PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK). *Journal of Open Source Software*, *4*(37), 1450. https://doi.org/10.21105/joss.01450

Team, G. L. (2020). *Introduction to relu activation function*. Retrieved 20th May 2022, from https://www.mygreatlearning.com/blog/relu-activation-function/

Xifeng Gao, M. T., Wenzel Jakob & Panozzo, D. (2017). *Robust hex-dominant mesh generation using field-guided polyhedral agglomeration*. Retrieved 29th May 2022, from https://cims.nyu.edu/gcl/papers/Robust-Meshes-2017.pdf

Zienkiewicz, O. C. & Taylor, R. L. (2000). *The finite element method, volume 1, the basis*. Butterworth-Heinemann.

Appendices

A GitHub Repositories

Project	Link
Simple FEM Solver, Hex8	github.com/Bragela/SolidFEM_BrickElement.git
SolidFEM, Hex20	github.com/Bragela/SolidFEM_20_node_hex.git
Fully connected NN, 2D	github.com/Isakflobodal/FullyConnectedNN_2dMesh.git
Unet, 2D	github.com/Isakflobodal/Unet_2dMesh.git
Fully connected NN, 3D	github.com/Isakflobodal/FullyConnectedNN_3dMesh.git
Encoder + Fully connected NN, 3D	github.com/Isakflobodal/Encoder-FullyConnectedNN_3dMesh.git
FEA pred, cantilever, ML	github.com/Bragela/FEM_cantilever.git
FEA pred, connection, ML	github.com/Bragela/FEM_connection.git
Rotational Stiffness pred, ML	github.com/AleksElm/RotStiff_ML.git

Table A.1: GitHub Repositories

B Videos

Filename	Description of video and link
Simple_FEM_Solver_gh.MP4	This video showcases Case Study 1, with the use of the Simple FEM Solver in Grasshopper. - Link: youtu.be/uARz1IRsHSY
SolidFEM_gh.MP4	This video showcases Case Study 2, for the arch beam, with the use of the SolidFEM in Grasshopper. - Link: youtu.be/os2tD25sbWM
pygmsh_continuity.MP4	This video illustrates discontinuity between the contour and the mesh when using the Pygmsh package. - Link: youtu.be/3Sv89Rf17Ak
pyvista_continuity.MP4	This video illustrates a continuous contour and mesh relationship. Here the the Pyvista package is used. - Link: youtu.be/jEQoJ6Wyiww
3D_DF_training.MP4	This video illustrates the development of the DF during training for the encoder + fully connected neural network in Case Study 4. - Link: youtu.be/Uxp_MF8D1oE

Table B.1: Videos with description and link to youtube

Aakre, Bødal, Elmer

Structural Analysis in Algorithms Aided Design

NTNU
Norwegian University of
Science and Technology