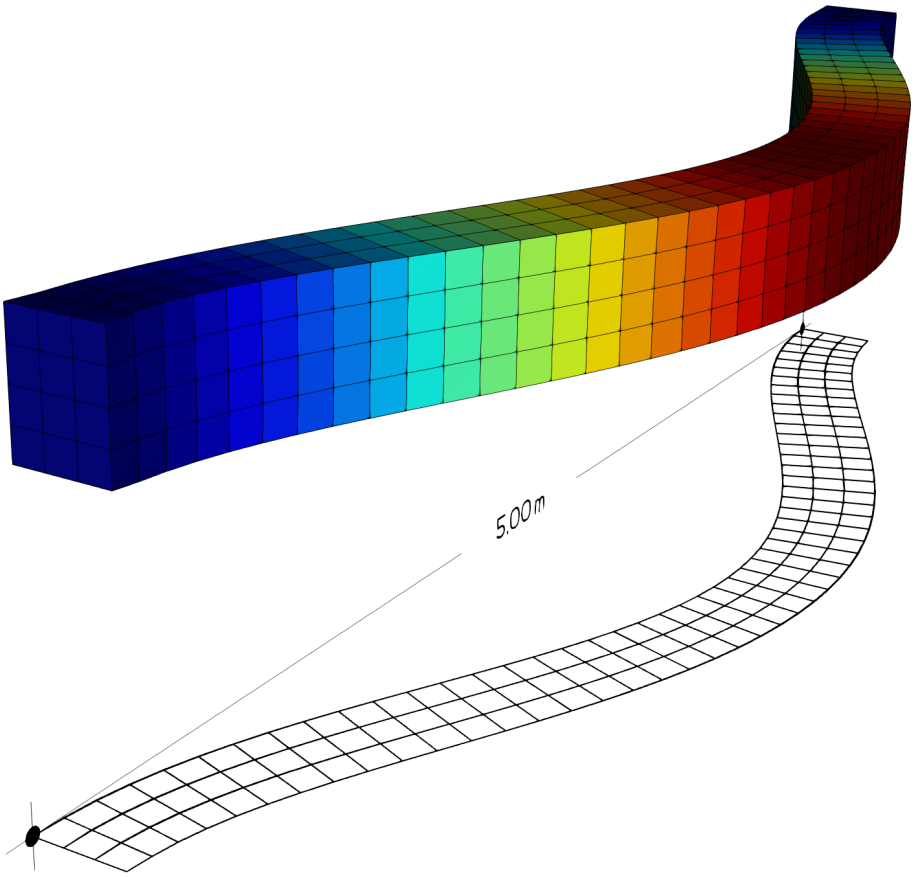


DEVELOPMENT OF VISUALIZATION FUNCTIONS IN CALFEM FOR PYTHON

ANDREAS ÅMAND

Structural Mechanics

Master's Dissertation

DEPARTMENT OF CONSTRUCTION SCIENCES

DIVISION OF STRUCTURAL MECHANICS

ISRN LUTVDG/TVSM--22/5256--SE (1-95) | ISSN 0281-6679

MASTER'S DISSERTATION

DEVELOPMENT OF VISUALIZATION FUNCTIONS IN CALFEM FOR PYTHON

ANDREAS ÅMAND

Supervisor: Dr **JONAS LINDEMANN**, Division of Structural Mechanics, LTH | Lunarc.

Examiner: Dr **OLA FLODÉN**, Division of Structural Mechanics, LTH.

Printed by V-husets tryckeri LTH, Lund, Sweden, March 2022 *(Pl)*.

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

Abstract

Visualizing results is an important part of Finite Element (FE) modeling. Many tools exist for visualizing these results, they are often part of complete software packages for FE analysis. These packages often rely on the user being familiar with FE analysis, making them unsuitable for use in teaching the FE Method.

CALFEM (Computer Aided Learning of the Finite Element Method) is an FE toolbox developed at LTH with an emphasis on teaching. It enables the user to implement the steps that otherwise are done behind the scenes in commercial FE software. CALFEM exists in two variants today, the original one implemented in MATLAB, while a Python version is continuously being developed in parallel. The main aim of this thesis is to develop visualization tools for the Python version of CALFEM. This will be integrated into the existing toolbox, allowing users to visualize results from calculations and aiding in the understanding of the FE Method.

To identify the needs for visualization, a study was conducted of the current visualization tools available in CALFEM. Current visualization tools only allow for visualizing of 2D problems. While CALFEM has support for many 3D elements, visualizing these in full 3D is not currently possible. The focus for this thesis is therefore on 3D visualization. Since using CALFEM requires manual scripting, visualizing geometries and meshes before solving the problem is very helpful, and emphasis is therefore put on visualizing all steps in the FE process. Another study was conducted to understand the visualization needs using CALFEM at relevant departments at LTH.

Python has an extensive amount of libraries for visualizing scientific data structures. A third study was conducted to find suitable libraries for implementing the identified visualizations in CALFEM. As a result of this study, a library called Vedo, based on the Visualization Toolkit (VTK) was chosen.

Using the libraries and results from the two first studies, several visualization functions for several different element types were implemented. Care was taken to implement these as seamlessly as possible into the Python version, following the same principles, syntax, and naming of existing functions. Functionality for importing data for visualization from the MATLAB version was implemented, along with export to VTK-files. The export functions allow for visualization of the results in ParaView, a more advanced open-source visualization tool.

Sammanfattning

Visualisering är en viktig del av finita elementmodellering (FE). Det finns många verktyg för att visualisera dessa resultat, dessa tillhör ofta mjukvarupaket för FE-analys, vilket gör att de är mindre lämpliga att använda för att lära ut FE-metoden.

CALFEM (Computer Aided Learning of the Finite Element Method) är ett finita element paket utvecklat vid LTH med fokus på undervisning. Verktyget ger en användare möjlighet att implementera de steg som annars är dolda i kommersiell FE-programvara. Det finns idag två varianter av CALFEM, orginalversionen i MATLAB, och en Python-version som utvecklas parallellt. Huvudmålet med detta arbete är att utveckla visualiseringsfunktioner till Python-versionen av CALFEM. Dessa funktioner ska kunna integreras med existerande funktioner, för att ge användare möjligheten att visualisera resultat från beräkningar och underlätta förståelsen för FE-metodik.

För att identifiera visualiseringsbehovet gjordes en studie av nuvarande visualiseringsfunktioner i CALFEM. Nuvarande visualiseringsfunktioner möjliggör endast visualisering av tvådimensionella problem. CALFEM stödjer dock många 3D-element, men att visualisera dessa dessa i tre dimensioner är inte möjligt. Fokus för arbetet har därför legat på visualisering av 3D-problem. Eftersom att CALFEM kräver att användaren skriver kod, är visualisering av geometri och mesh innan beräkning värdefullt. Vikt har därför lagts vid att kunna visualisera dessa stegen i FE-analysen. En undersökning gjordes för att utreda behovet av visualisering med CALFEM hos de avdelningar på LTH som använder CALFEM.

Python har en omfattande mängd bibliotek för att visualisera vetenskaplig data. En tredje studie genomfördes för att finna lämpliga bibliotek som kan användas för att implementera de identifierade visualiseringsbehoven i CALFEM. Resultatet av denna studie var att använda Vedo-biblioteket som är baserat på Visualization Toolkit (VTK).

Men hjälp av biblioteken samt resultaten av de två första studierna implementerades flera visualiseringsfunktioner för ett antal elementtyper. Dessa har implementerats med hänsyn till att kunna fungera bra med Python-versionen av CALFEM. Samma principer, syntax och namngivning av funktioner följs. Funktionalitet för att importera data för visualisering från MATLAB-versionen har implementerats, samt export till VTK-filer. Exportfunktionaliteten möjliggör visualisering av resultat i ParaView, ett mer avancerat visualiseringsverktyg baserat på öppen källkod.

Acknowledgements

This dissertation is the result of work carried out during the Autumn semester of 2021 at the Division of Structural Mechanics, Faculty of Engineering, Lund University, Sweden.

I would like to thank my supervisor Jonas Lindemann for help throughout the project, for suggesting the idea and helping with coding issues on a short notice. I would also like to thank the staff at the Divisions of Structural Mechanics and Solid Mechanics for their input.

Finally, I would like to thank my family and friends for their persistent support throughout my education.

Lund, January 2022

Andreas Åmand

Notations and Symbols

Code

⌋	Line break
↪	Line continues
...	Discontinuous code

Abbreviations

FE	Finite Element
CALFEM	Computer Aided Learning of the Finite Element Method
VTK	The Visualization Toolkit
OpenGL	Open Graphics Library
API	Application Programming Interface
Dim.	Dimensions
Def.	Deformed
Undef.	Undeformed
Disp.	Displacement
El.	Element
DoF	Degrees of Freedom

Contents

1 Introduction

1.1 Background

In teaching engineering subjects, understanding physical phenomena are important for students. Providing good visualizations of these can aid in understanding when introduced to a new subject. With modern graphics hardware, many new improved visualization techniques are possible. In a Finite Element (FE) analysis, visualization is usually done mostly at the final stage, when results are obtained and analyzed. There are however many steps along the way where visualizations are possible, see Figure 1.1.

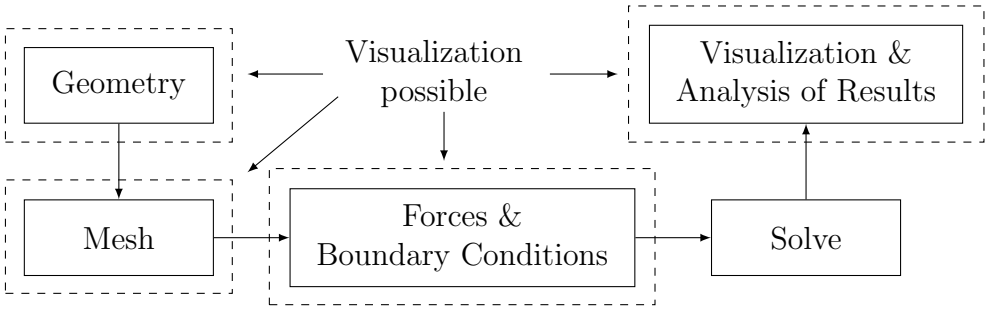


Figure 1.1: Typical Finite Element workflow & possible visualization stages

CALFEM is a FE toolbox for teaching developed at the Division of Structural Mechanics, LTH [1]. It was originally developed for MATLAB in the late '70s, today there are also versions available for other programming languages, including Python [2]. Both MATLAB and Python versions implement functionality for visualizing results in 1-2 dimensions, visualizing 3D models is however lacking. 3D problems can be solved and visualized using 2D plots, there is however no functionality for visualizing in 3D.

To aid in understanding the FE method, and results from FE analysis in CALFEM, tools for visualization need to be improved and extended. Some visualization tools in the Python version are also missing from the MATLAB version.

Python is an open-source high-level programming language. It has been highly adopted for scientific programming [3], and therefore many libraries have been created for different scientific applications. There are many options for visualization available in Python.

The current functionality for visualization in CALFEM for Python also needs improved error handling, as it gives the user little feedback if something goes wrong.

For more complex problems, exporting to more powerful visualization tools is often required. In the Python version, exporting for use with the visualization tool ParaView is possible, but requires manual scripting by the user.

1.2 Aim & Objective

The main aim of this project is to improve visualization functionality in CALFEM. This can be broken down into a few parts. The first part is considering what visualizations functionality currently exists in CALFEM for Python.

The next part is to add visualization functionality with a focus on teaching and education. Part of this is considering the visualization needs at departments using CALFEM at LTH. Adding functionality here means adding support for more types of elements and visualizations for these elements.

In the final part, functionality for importing/exporting data is to be improved in the Python version. This accomplishes multiple things, making the Python version more versatile. Importing/exporting should be handled in few steps and should be easy to use. Export to ParaView and possibly other visualization tools are also implemented. Also, as the MATLAB version of CALFEM is more widely used, being able to import MATLAB data is therefore useful for making the visualization tools accessible and possibly used more broadly at LTH. Adding functionality for importing MATLAB data and exporting data will also make the Python version a bridge between MATLAB and more powerful tools.

Before developing any additional functionality for visualization, some central questions that will be in focus are:

- What can be visualized using the current functions in CALFEM for Python?
- What additional visualization functionality is needed in CALFEM for Python?
- What Python library/libraries are suitable to implement the above?

1.3 Limitations

Since this thesis in part involves identifying visualization needs, the additional functionality to be implemented is mostly limited to these. This also means visualizations already implemented will be overlooked. Possible exceptions to this are if some visualizations are supported for only some elements and can therefore be expanded upon.

1.4 Method

To get an understanding of what functionality for visualization is available in CALFEM for Python, current tools for visualization will be examined. Results from this study will partially decide where the focus is put in terms of adding visualization tools. The study of current visualization tools will be supplemented with a survey of visualization needs in CALFEM at relevant LTH departments. This is done to get a better understanding of what visualization is relevant for teaching courses.

When the initial two studies have finished, a survey of Python libraries for visualization will be done. This is to get a better understanding of what existing libraries would be useful for adding functionality for visualizing in Python. Since it is known 3D visualization is lacking in CALFEM, Python libraries with support for 3D will be the main focus. Various considerations regarding needs in CALFEM will be listed in the study. When deciding on what tools to be used, these considerations will be referenced in order to base the choice on what is needed for CALFEM.

When functionality is lacking in CALFEM and implementation libraries to use have been chosen, additional tools for visualization will be developed. When developing additional tools for CALFEM, care will be taken to add as little additional scripting as needed by the end-user. This is because of the strong focus on usability and teaching aspects of the project.

Exporting of results for viewing in ParaView (and possibly other tools) will be improved by implementing functionality for automatic handling of this. This functionality is considered central to the project, and important evaluation criteria when deciding what Python libraries to use for development.

Importing of MATLAB data will be implemented using functions for reading '.mat' files. This means a MATLAB user will only need to save workspace variables and import them into CALFEM for Python, to visualize them.

1.5 Disposition

Chapter 2 contains the surveys of the current visualization tools in CALFEM for Python and the survey of visualization needs. Following these, the results from the survey to find suitable Python libraries for visualization is presented, concluding with the aspects considered. Chapter 3 describes important aspects for the chosen libraries, along with basic usage, also further expanding on aspects of why they were chosen. Chapter 4 describes how the implementation of functions into CALFEM is accomplished. Most functionality of the tools is explained here, some choices made during development are also presented, along with issues that caused them. Chapter 5 contains five examples of different visualizations possible with the new functionality, along with basic explanations of how they are done. Many aspects of the functionality implemented in Chapter 4 are showcased here. Chapter 7 contains conclusions regarding what has been implemented and what it can be used for. Chapter 6 discusses further aspects of the development and the final results from it. Chapter 8 suggests possible ways to expand upon the functions developed.

This is followed by references, and an Appendix A containing a manual for using the functions. This Appendix contains more detailed descriptions (including relevant source code) for examples in Chapter 5. To get a brief overview of what has been implemented, over-viewing this Appendix (especially A.7) is suggested before reading Chapter 4 and onward. The MATLAB code used for the solid example in Chapter 5.4 is extensive and can be found in Appendix B. The form used to conduct the survey in Chapter 2.2.2 can be found in Appendix C.

2 Existing visualization tools

In this Chapter, the three previously mentioned studies are presented. In Chapters 2.1 and 2.2, the current state of visualization in CALFEM for Python is presented. The survey of visualization needs is presented in Chapter 2.2.2. Chapter 2.3 presents some considerations for the study of existing Python libraries to use in Chapter 2.4. The results from this study is then presented in Chapter 2.5.

2.1 Visualization in CALFEM for Python

CALFEM for Python currently contains two modules for visualization, `vis` and `vis_mpl` [2]. These modules are based on the Python libraries Visvis and Matplotlib respectively. Some of the functions in these modules overlap and some are unique to each module. A reference for what these modules contain are given in Table 2.1.

Function	Dimensions	`vis_mpl`	`vis`
`draw_geometry`	2D	✓	✓
`draw_mesh`	2D	✓	✓
`draw_displacements`	2D	✓	✓
`draw_element_values`	2D	✓	✓
`draw_nodal_values`	2D		✓
`draw_nodal_values_contour`	2D	✓	
`draw_nodal_values_contourf`	2D	✓	
`draw_nodal_values_shaded`	2D	✓	
`eldraw2`	2D	✓	✓
`eldisp2`	2D		✓
`eliso2`	2D		✓
`elval2`	2D		✓

Table 2.1: Existing main functions for visualization in CALFEM for Python

The current focus of these modules is on 2D problems. The VisVis module will not be supported in future versions of CALFEM in favor of Matplotlib. This module will therefore form the basis for the 2D visualization in CALFEM moving forward. Something to note from Table 2.1 is that some functions exist in both modules. In addition, some of the functions also somewhat overlap in functionality. Functions containing `_nodal_` are for instance plotting nodal data but implemented in slightly different ways.

What functions exist for visualizing gives a limited overview of actual functionality. For a further overview of what the functions can do, a breakdown based on different element types is necessary. This is done in Table 2.2, based on both `vis` and `vis_mpl`.

Element			Mesh			Scalar		Ex. function in CALFEM
Type	Dim.[1]	Geo.[2]	Def.[3]	Undef.[4]	Disp.[5]	El.[6]	Nodal	
Spring Element								
Spring	1D	✓	✓	✓	✓	✓	✓	`spring1e`
Bar Elements								
Bar	2D	✓	✓	✓	✓	✓	✓	`bar2e`
	3D	-	-	-	-	-	-	`bar3e`
Flow Elements								
Flow	2D	✓	×	✓	×	✓	✓	`flw2te`
	3D	-	×	-	×	-	-	`flw3i8e`
Solid Elements								
Solid	2D	✓	✓	✓	✓	✓	✓	`soli3e`
	3D	-	-	-	-	-	-	`plante`
Beam Elements								
Beam	2D	✓	✓	✓	✓	-	-	`beam2e`
	3D	-	-	-	-	-	-	`beam3e`
Plate Elements								
Plate	2D	✓	-	✓	-	-	-	`platre`

Table 2.2: Breakdown of visualization based on elements in CALFEM for Python.
Notation: '✓'-supported, '-'-unsupported, '×'-not possible.
[1]Dimensions, [2]Geometry, [3]Deformed, [4]Undeformed, [5]Displacement, [6]Element.

2.2 Missing functionality in CALFEM for Python

Visualizing geometry is currently not supported in 3D. Adding functionality for this would allow verifying that a 3D geometry is correct. After meshing, it's also useful to be able to verify that the mesh corresponds to the geometry.

After an FE problem is solved, the mesh can be displayed in an undeformed or deformed state. This is done in order to see the shape and/or get nodal/element values from the model, typically at certain critical points in the model. In 2D, this can be done using various functions from Table 2.1 in Chapter 2.1. This is not supported for 3D models, which means that the visualization part of the workflow is not currently possible in CALFEM.

2.2.1 Import & export

Something of note regarding CALFEM, in general, is that the most widely used version of the toolbox is the MATLAB version. This toolbox was originally written for it and most courses at LTH still use the MATLAB version. In the interest of visualization tools in CALFEM for Python being as widely used as possible, functionality for visualizing data from the MATLAB version is useful. This is currently not supported in the Python version.

For visualizing larger datasets and more complex problems, it is useful to export to more powerful tools for visualization. ParaView is a sophisticated tool for visualizing

many types of scientific data (not a Python library). The current version of CALFEM for Python allows for manual export to VTK to use in ParaView. This is however not very easy to do and relies on the user saving the data correctly. To aid in using CALFEM for a wider range of problems, implementing better functionality for exporting to the VTK format is useful. This would also allow the Python version to act as a bridge between the MATLAB version and ParaView.

2.2.2 Survey of visualisation needs

To get some input on what tools would be useful in teaching using CALFEM, a survey was sent to staff at LTH departments (see Appendix C). The survey is a digital PDF with a form filled out digitally. It was formulated to be as short and simple as possible, so that staff members would check what they wanted to visualize, also allowing for additional comments if needed. The survey included what types of elements would be useful and for those elements, what types of visualization would be useful for them. The intention of doing a short survey was to get as many responses as possible, and therefore fairly broad input.

A total of 5 responses from the departments of Structural and Solid Mechanics was received. Because of the low amount of responses, the visualization tools cannot be entirely based on this survey. The survey is still useful for input regarding what is useful for specific courses and where the focus should be put. It was formulated based on what types of elements are already implemented and types of visualizations possible for these elements CALFEM.

Four staff members were currently involved in courses using some version of CALFEM, and all participants would consider using it more for teaching if visualization tools were improved. Notable results from the survey are that various 2D visualizations are the most widely asked for. Also, animations of deformations and responses of structures were considered useful by 4 staff members. One response also commented that most visualizations in the survey should be useful for courses at the departments. An overview of results is shown below in Table 2.3.

Element	1D	2D	3D
Spring	3	-	-
Bar	4	3	1
Flow	2	1	-
Beam	-	5	2
Solid	-	3	1
Plate	-	1	-

(a) Breakdown by element types

Visualization	1D	2D	3D
Undef. mesh	4	5	3
Def. mesh	4	5	3
Displacements	4	5	3
Scalar values	4	4	2
Section forces	3	4	2
Stresses	4	4	2
Principal stresses	-	3	1
von Mises stresses	-	3	1
Contour plots	-	3	1
Isolines	-	1	-

(b) Breakdown by visualization types

Table 2.3: Results from survey

For visualizations of 1D elements, nearly every type of visualization possible for these elements are asked for. The same is true for 2D elements. Here, a more clear breakdown by element type can be seen. Some of the visualizations for 1D and 2D elements are already possible in CALFEM for Python. However, these can be improved or integrated into developed tools. The survey acts as guidance on what functionality for 1D and 2D can be improved.

From the survey it's clear that for visualizations of 3D elements are least useful, some basic functionality for mesh and deformations are asked for. Also, stresses and scalar values appear to be useful. Since the scope of the project includes a focus on 3D visualization. This survey will act as some guidance for implementing this functionality in CALFEM for Python, by attempting to implement the most requested visualizations using the library/libraries chosen.

As bars, beams, and solids were requested for 3D visualizations, these will be prioritized. Undeformed and deformed meshes are essential for these. Displacements, scalars, section forces (for beams) and stresses will also be prioritized. When these are implemented, essentially everything in Table 2.3 will be attempted to try and make the functionality as complete as possible, supporting multiple elements, dimensions, and visualization types for these. Animations will also be attempted.

2.3 Considerations for Python libraries

Before moving on to surveying the Python visualization landscape and finding suitable libraries to use for CALFEM, some criteria need to be considered. To find relevant libraries, there are many considerations made, some more important than others.

Having functionality for visualizing both 2D and 3D is essential. This is because some visualizations for 3D elements are still done in 2D because of the nature of plotting some physical phenomena. An example of this could be 3D beam elements, which still need sectional properties plotted on a 2D plane. Fortunately, most visualization libraries that support 3D also support 2D, so focus can be put on 3D functionality.

Since CALFEM is an FE toolbox, focus is put on visualization of FE analysis. Any library for visualizing 3D objects will be able to visualize FE geometry. However, for visualizing meshes, deformations, contours, and animations, this functionality is specifically required in the library (or underlying libraries). To evaluate what the libraries can do, their documentation and examples are studied. Consideration is also taken to whether the libraries are utilized by existing FE tools, as this gives additional credibility.

Some considerations are made concerning the development process. For instance, good and extensive documentation and ease of use in Python are evaluated. Since some libraries are difficult to use, implementing different elements and visualizations can hinder development. To allow for implementing as many element types and visualizations as possible, this is an important aspect. This is also useful in further development of the visualization tools down the line.

Some consideration is also taken to how established the package is, how well maintained it is and who is maintaining it. Python development is moving ahead rapidly and many libraries get updated often. These updates can break dependencies, cause errors and make some functionality unusable. If a package is not well maintained or in an abandoned state, this can be a possible problem. Considerations for how many dependencies a library has are also made for the same reason, the fewer the better as many dependencies updates regularly, often causing problems with reliability.

To simplify the use of the tools for the end-user, a user interface is helpful. This allows for implementing functions for manipulating the model, importing and exporting easily for example. A decision was made early on to utilize Qt via the PyQt5 (Python Qt) binding for Python. This is a versatile, open-source, and proven cross-platform solution built on C++ [4]. It also interfaces with many Python libraries, including visualization libraries.

2.4 Python-libraries for visualization

Matplotlib has extensive functionality for 2D plotting. For some functionality within the scope presented, it makes sense to use it. For 3D visualization, efficient 3D rendering is needed. For this purpose, Matplotlib is limited. It can generate 3D plots, however rendering more complex objects in 3D, is not efficient.

To overview the available tools for visualization that could be utilized, the PyViz [5] project is a good resource. The project indexes most visualization tools built using Python, by types of visualization tools and needs for the end-user. All tools they reference are open-source libraries with permissible licenses. The existing tools utilized by CALFEM for Python, Matplotlib, and Visvis are also indexed here. Scientific visualization tools, mainly for 3D visualizations, are categorized under SciVis libraries. SciVis is an abbreviation commonly used in the Python developers community referring to visualizing any scientific data [6].

Most SciVis libraries are based on OpenGL (Open Graphics Library), an open multi-purpose graphics standard for hardware-accelerated graphics. Some examples of SciVis libraries utilizing OpenGL are VTK (The Visualization Toolkit), yt, VisPy, and glumpy [5].

2.4.1 The Visualisation Toolkit - VTK

VTK is maintained by Kitware inc. [7] and used by the ParaView application [8]. It was developed in the '90s [9] and has become an established and well-maintained library, and therefore widely used. However, it is relatively difficult to use, not designed with usability in mind, and it has a complex data processing pipeline [10] (chain of processing, see Figure 3.1). It is intended for developing powerful visualization applications. It has very high computational efficiency thanks to its compiled C++ library [9]. See Table 2.4 for an overview.

Pros:	- Capable of every type of visualization required for this project
	- Extensive well written documentation available
	- Support for PyQt5
Cons:	- An older library with some functions only available in Python 2
	- Most documentation is written for the C++ version of VTK
	- Complicated scripting.

Table 2.4: Overview of VTK

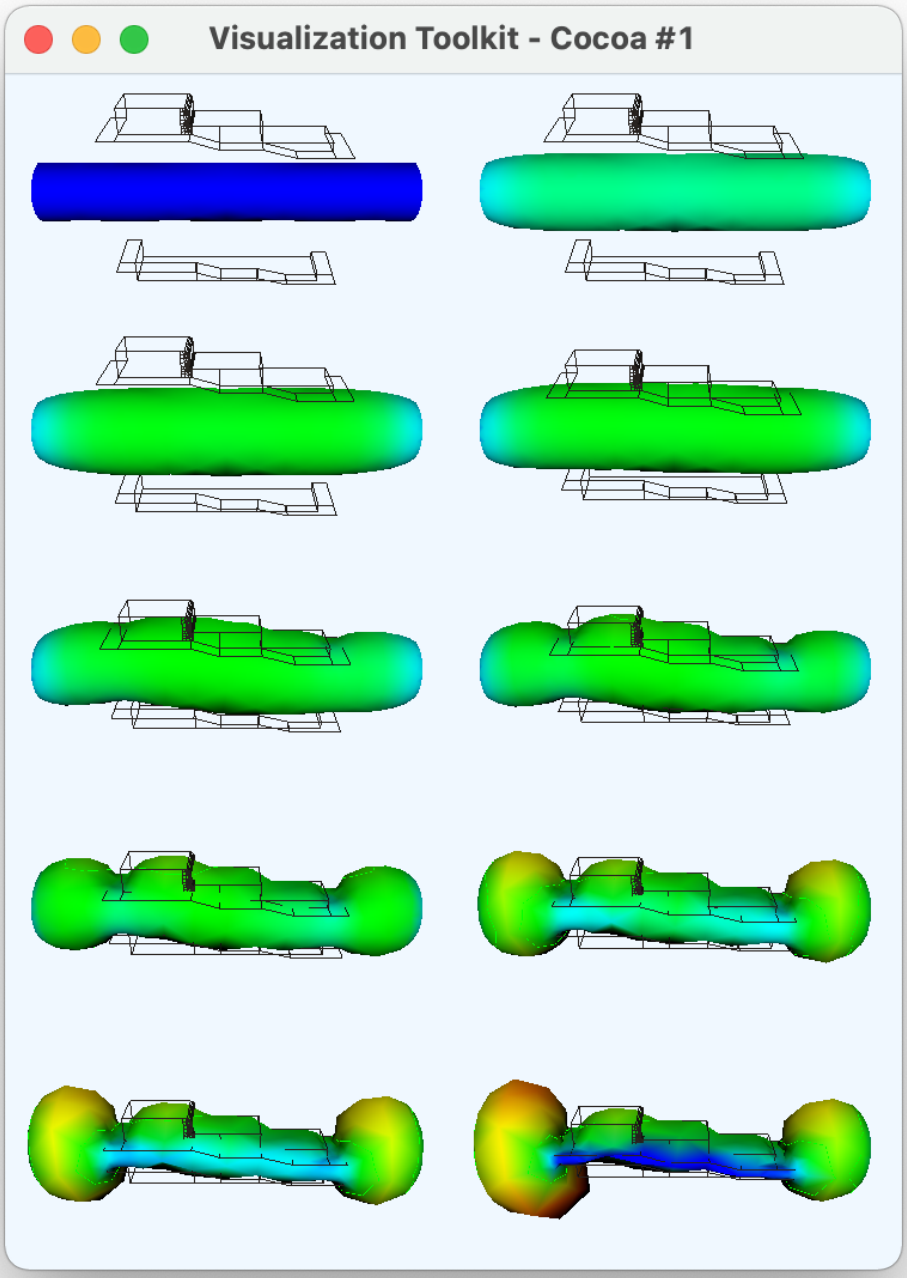


Figure 2.1: VTK example provided by VTK [7]

VTK is a strong candidate for this project, it can be used for visualizing FE-analysis. Kitware provides examples of many types of visualizations, among them a visualization of ten steps in a nonlinear-elastic analysis [7]. This example can be seen in Figure 2.1. Many open-source FE applications rely on VTK for visualization, for example, MOOSE (Multiphysics Object-Oriented Simulation Environment) [11].

2.4.2 Mayavi

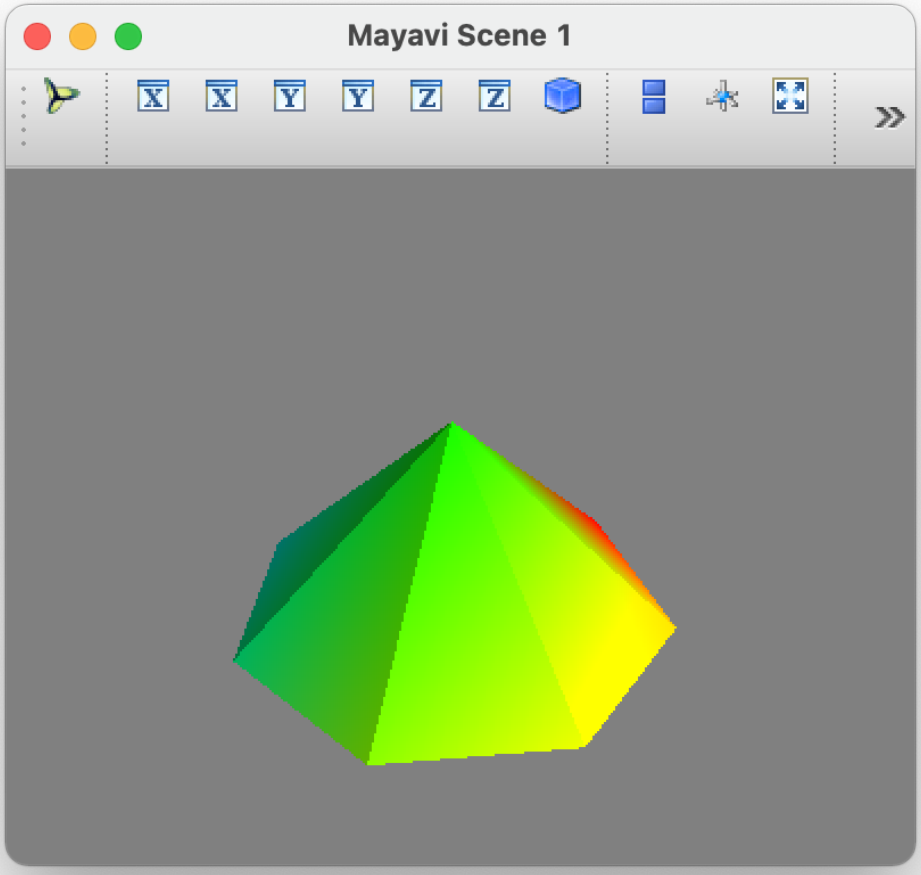


Figure 2.2: Mayavi example

Another example of a widely used application/library utilizing VTK is Mayavi. It is maintained by Enthought inc. [12] and meant to extend VTK to be more versatile, being both scriptable in Python and giving users a built-in application to work with visualizations. It also supports Jupyter notebooks, a useful tool for presenting Python code and running it in a webpage [3]. Because of its versatility, it depends on many other libraries [12][13], potentially making it difficult to install and maintain a working installation. An example of a visualization using Mayavi can be seen in Figure 2.2, note the built-in user interface at the top. See Table 2.5 for an overview.

Pros:	- Capable of every type of visualization required for this project
	- Very flexible, with support for PyQt5 & Jupyter
Cons:	- Not a lot of documentation available
	- Difficult to install and use because of many dependencies

Table 2.5: Overview of Mayavi

2.4.3 PyVista & Vedo

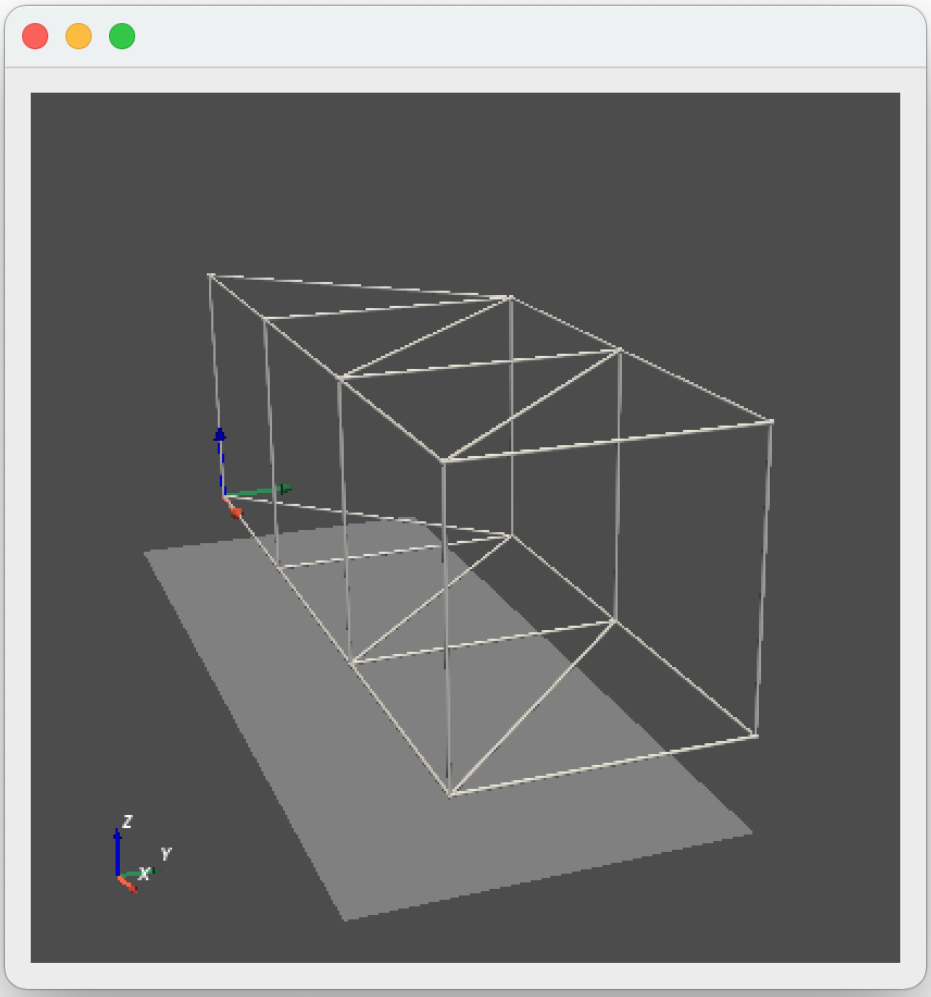


Figure 2.3: PyVista example

PyVista and Vedo are also part of the SciVis category of the PyViz repository. They intend to maintain the powerful functionality of VTK while making it easier to use and seamlessly integrate with it [14][15]. Both are developed by individuals working in many different research fields. Both have modules integrating with the FEniCS (Finite Element Computational Software) project [16], which is an open-source FE application. This makes them good candidates for use in FE visualizations. Figures 2.3 and 2.4 show examples of what visualizations using them can look like. See Table 2.6 for an overview of both, as they are very similar.

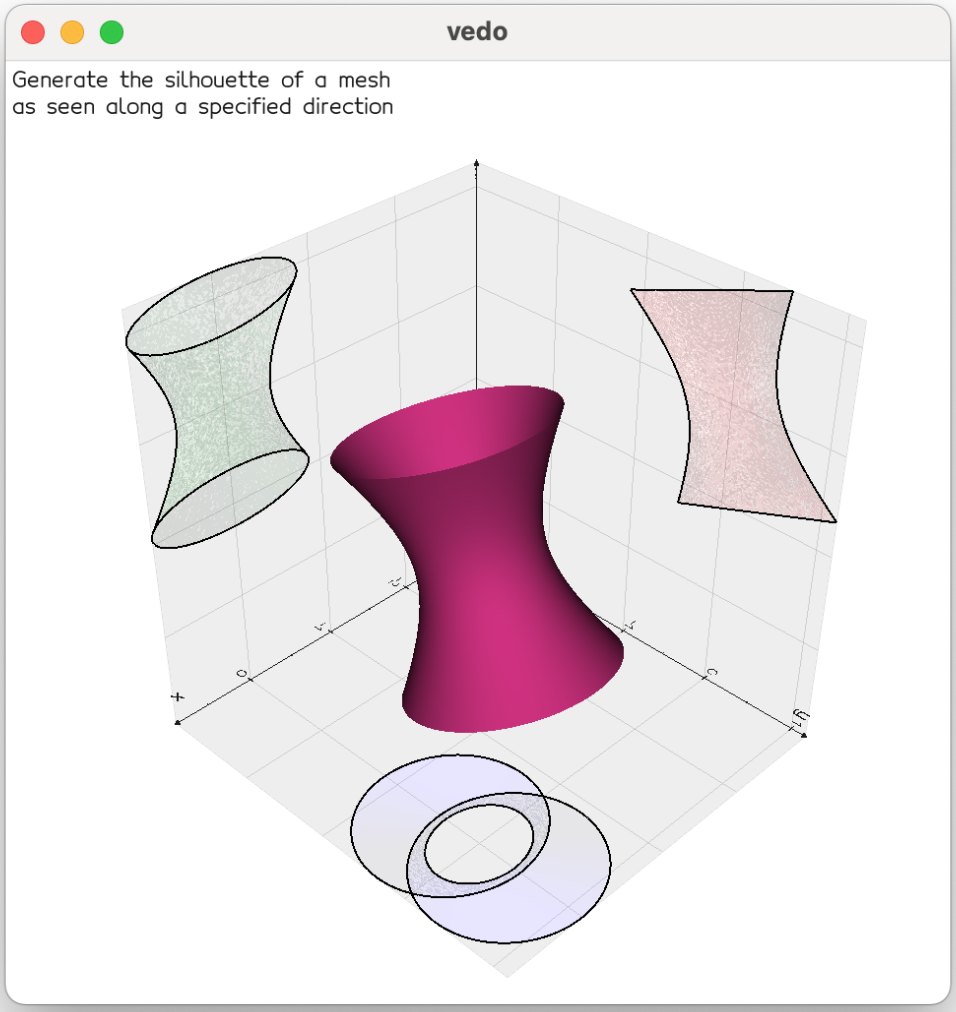


Figure 2.4: Vedo example provided by Vedo [14]

Pros:	- Utilizes the strengths of VTK
	- More scripting friendly than VTK & Mayavi
	- No dependencies other than VTK for base functionality
	- Supports export to VTK file format
Cons:	- Not full implementations of VTK
	- Maintained by individuals/small teams

Table 2.6: Overview of PyVista & Vedo

2.4.4 Polyscope

Polyscope is a library for rendering 3D models interactively. Interaction means a user can manipulate the model live to change the visualization or get information from the model. It is compatible with Vedo and allows for extended and simple interactions.

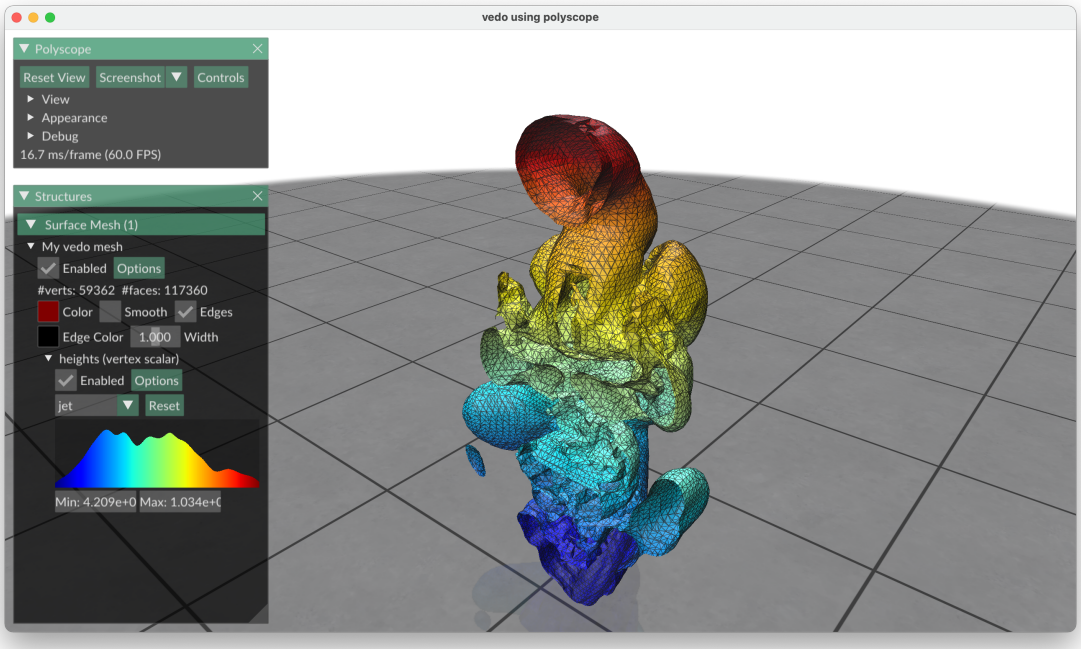


Figure 2.5: Polyscope example provided by Vedo [14]

Polyscope has two major drawbacks. Since it cannot use Qt, it means regular menus for interacting with the model and other possible features, are not possible. If used for visualizations in CALFEM, interaction would largely be limited to using Polyscope's built-in menus. This imposes many limitations. The greatest limitation is scripting interactions with anything not related to the rendering, which is often not possible. This is because Polyscope does not use underlying VTK classes when rendering, not permitting any manipulation of the model apart from the menus it implements as standard. This makes Qt more suitable for interaction, compared to Polyscope. Figure 2.5 shows an example using Polyscope, note the menus for interaction on the left side.

Polyscope does not render using VTK, instead relies only on OpenGL [17]. The model would be created using VTK but rendered using different classes. If any VTK-based library was used, the underlying classes for rendering would be accessible for scripting. Polyscopes implementation has many drawbacks in itself. In general, the compatibility between Vedo & Polyscope is not very good at this time. Even the examples using Polyscope provided by Vedo open two different windows, one showing a rendering using Polyscope, one using Vedo. This suggests a poor implementation of Polyscope-Vedo interaction. See Table 2.7 for an overview.

Pros:	- Extended interactivity compared to Vedo - Better looking renderings than Vedo - Uses few computational resources	**Cons:**	- No PyQt5 - No VTK rendering

Table 2.7: Overview of Polyscope

2.5 Useful libraries for CALFEM for Python

All of the libraries mentioned so far in this Chapter have been tested to some extent, mostly VTK, MayaVi, PyVista, Vedo and Polyscope. When evaluating them beyond cited sources (documentation and academic papers), drawing simple objects was attempted. Some manipulation of these via coloring/interaction was used for testing as well. This is done in order to get an understanding of how the tools are intended to work and how suitable they are for CALFEM.

VTK was first chosen for the project as it is powerful and well established. This was before Vedo/PyVista had been fully tested. Other OpenGL-based libraries cannot be considered as useful for visualization in CALFEM. The reason for this is that exporting to more powerful visualization tools is part of the requirements for the libraries. ParaView is an excellent example of a tool that fulfills the criteria for this. Since it's based on VTK, building the CALFEM visualization tools on VTK is beneficial for implementing export functionality. Other OpenGL-based libraries (like yt, VisPy etc) were rejected for CALFEM, in part due to a lack of functionality for exporting to VTK format.

After testing VTK alongside CALFEM, it became clear that the decision to base the visualization on VTK was beneficial. It allows for building an intuitive class structure in Python for rendering 3D graphics. However, since it is a library with fairly experienced C++ programmers in mind, it is difficult to use its full potential due to its complexity. It became clear that using standard VTK was hindering the development of functionality for different elements and types of visualizations for these in CALFEM. Since there are multiple libraries built upon VTK, meant to simplify development while maintaining functionality, a decision was made instead to use a suitable VTK-based library for further development. Further surveying libraries suitable for use with CALFEM were therefore made. Mayavi was decided against using because of its heavy reliance on external libraries. From testing Mayavi, it was also found to be too unstable. This could in the end make it difficult to use for any staff/student using CALFEM.

When further researching VTK-based libraries to use, PyVista and Vedo stood out. Both are meant to simplify the use of VTK [14][15]. They were both found to be easy to work with to develop visualization tools. In the end, Vedo proved to have the best workflow suitable for CALFEM development. Because of its class structure, handling data for visualizations worked very well, it proved to be very flexible. It is also easy to interact with the VTK-classes and adding/removing objects to the renderer is very efficient in code.

Because Vedo was chosen, further expanding upon it using Polyscope was considered. Since Polyscope does not support PyQt5 or VTK rendering however, it was quickly abandoned. using it would mean using a completely different rendering process than normally used in a VTK-based application. Both visualizing using developed functions, and exporting for use in ParaView are important. Using Polyscope would mean greater discrepancies between the model visualized in CALFEM, and the one exported to ParaView.

3 Development using VTK & Vedo

3.1 VTK

When developing an application using VTK, many classes are required to create a working application [7]. Some of these can be seen in Figure 3.1. To visualize something, first, a source object has to be created. This can be one of VTK's many existing source classes. If visualizing a cube is needed, then the class `vtkCubeSource` is used as a source object. This source object defines the basic geometry. Next, a mapper object is created. The mapper object is responsible for mapping the source object to a representation in 3-dimensional space. This means the mapper contains the coordinates of the source objects points (the eight corners of the cube). The source object is given as input to the mapper. Finally, an actor object is needed. The actor is a class for managing the rendering of an object in the scene. These stages are referred to as the visualization stage in VTK [7]. See Listing 3.1 for an example of how many actors can be created using a loop.

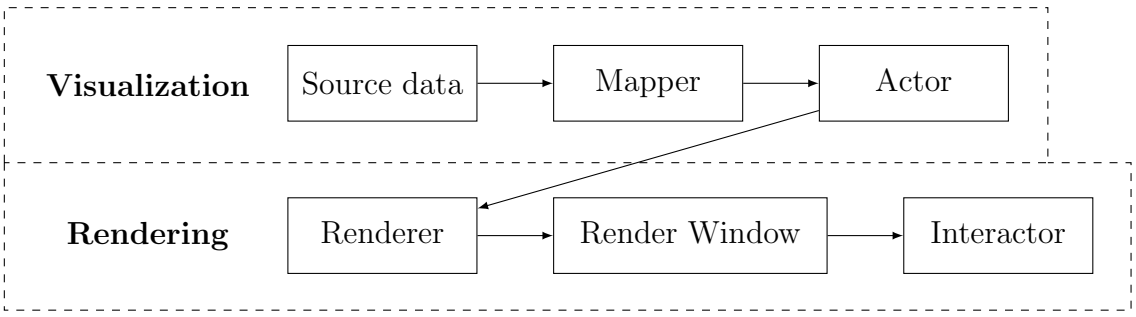


Figure 3.1: Basic VTK application structure

```
def set_geometry(edof,coord,dof):
    node_actors = []
    ncoord = np.size(coord, axis = 0)
    for i in range(ncoord):
        node = vtk.vtkSphereSource() # Get source
        node.SetCenter(coord[i]) # Set origin
        node.SetRadius(0.05) # Set radius

        mapper = vtk.vtkPolyDataMapper() # Create mapper
        mapper.SetInputConnection(node.GetOutputPort()) #
         ↪  Input source
        node_actors.append(vtk.vtkActor()) # Create actor
        node_actors[i].SetMapper(mapper)) # Input mapper
```

Listing 3.1: Creating actors using the `vtkActor`-class

After an actor is created, the classes related to rendering in Figure 3.1 are required. The process of generating an image on the screen is, in 3D graphics, refereed to as rendering [18]. This is done in a similar way to the visualization stage, where each class is created and used as input to the next. If a Qt window is used, a few extra steps are required before the renderer is created, see Listing 3.2 for an example. The renderer handles the actual rendering of the actor object (via OpenGL). The render window contains the rendered image and the interactor allows the user to interact with the model (rotate camera, manipulate objects, etc).

```
def vtk_initialize(self):
    # Load UI
    uic.loadUi("QtVTKMainWindow.ui", self)

    # Create container
    self.vl = QtWidgets.QVBoxLayout()
    self.vtkWidget = QVTKRenderWindowInteractor(self.frame)
    self.vl.addWidget(self.vtkWidget)

    # Create renderer & render window
    self.ren = vtk.vtkRenderer()
    self.renwin = self.vtkWidget.GetRenderWindow()
    self.renwin.AddRenderer(self.ren)
    self.iren = self.renwin.GetInteractor()
    self.iren.SetRenderWindow(self.renwin)

    # Setting frame
    self.frame.setLayout(self.vl)

    ...

    # Starting render
    self.ren.ResetCamera()
    self.iren.Start()
```

Listing 3.2: Creating the rendering classes in VTK

Developing an application that is usable in VTK requires the use of many more classes than so far mentioned in this Chapter, and subsequent method calls on these classes. VTK is heavily object-oriented [19] and has become very complex over time as very specialized classes have been developed [10]. For instance, a separate class instance is needed to transform an actor's position.

Since VTK is written in C++, a lower-level programming language than Python, its Python binding is also lower level and expressive compared to other Python libraries [20]. This has advantages in computational efficiency but makes developing visualization applications in VTK very complex [15]. A higher-level API would therefore be beneficial for implementing the required functionality. For this reason, Vedo was introduced into development.

3.2 Vedo & VTK

Vedo is a Python API (Application Programming Interface) above the VTK Python binding. It uses its classes for interfacing with VTK, resulting in a more simple developer pipeline. The class structure can be seen in Figure 3.2, along with the corresponding VTK classes.

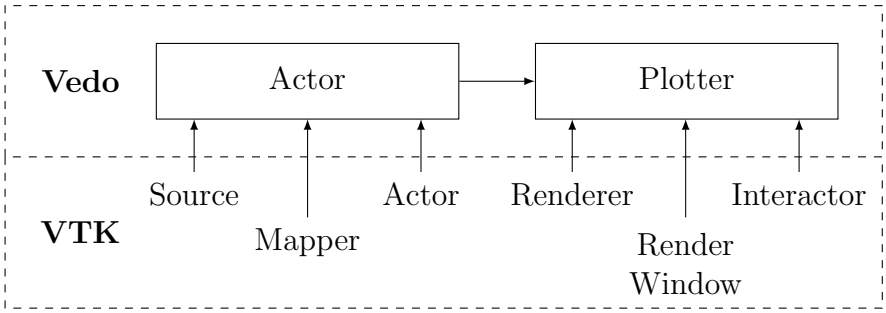


Figure 3.2: Vedo application structure & VTK classes used by them

There are many more VTK-classes utilized by Vedo that are simplified in the same manner as in Figure 3.2. To illustrate the differences in code, see Listing 3.3. Here, all the Vedo & VTK classes in Figure 3.2 are used. Actors, along with all classes needed for rendering, are created. The interactor is run, allowing the user to see the model and interact with it.

```
import vedo as v
class VedoMainWindow():
    def render(self,edof,coord,dof):
        plt = v.Plotter() # Create a plotter
        for i in range(np.size(edof, axis = 0)):
            plt.add(v.Sphere()) # Add actors
        plt.show(interactive=True) # Show plotter
```

Listing 3.3: Creating actors and a renderer in Vedo

Comparing the code in Listing 3.2 with the examples in Chapter 3.1, Vedo is very efficient in providing the same functionality as VTK with very little code. This allows for more efficient development as functions can be easily tested on the fly in Vedo, without setting up everything that VTK requires. At the same time, if access is needed to the underlying VTK-object, Vedo allows for this as well.

4 Implementation

Vedo was chosen as the Python library for the new 3D visualization functions in CALFEM. Different aspects of how it has been implemented will be shown in this Chapter. Many of the visualization functions also utilizes NumPy [21]. This is a library implementing most of the array features found in MATLAB [22]. Many implemented functions utilizing Vedo are referenced in this Chapter, for descriptions and usage of these, see Appendix A.

4.1 Geometry & Mesh

For visualising FE geometries, `draw_geometry` is implemented. This function requires the use of the geometry module [23] in CALFEM for Python. It supports rendering points, lines, and surfaces defined by the user when creating the geometry. For an example utilizing this function see Chapter 5.3. CALFEM also supports interactive editing of geometries in 2D [24], as this editor uses the existing geometry module in CALFEM, visualizing geometry from this editor should also be possible.

For visualizing FE mesh using Vedo, two functions have been implemented, `draw_mesh` & `draw_displaced_mesh`. These are similar in functionality, the main difference is the latter one takes a global displacement vector as input. Both require `Edof`, `Coord` & `Dof` matrices used in CALFEM [1] as input. `Edof` contains the element topology, each row represents the degrees of freedom for an element. The global degrees of freedom are stored in `Dof`, and the corresponding coordinates in `Coord`.

The two mesh functions also require an `element_type` variable. This determines the type of mesh to be visualized. Based on the element type and matrices given as input, each element is iterated through using a loop, and an actor is created to render it. This is done by checking the row in `Edof` for all the degrees of freedom and then finding the coordinates that correspond to those degrees of freedom. If for instance, a beam element is to be rendered and matrices in Listing 4.1 are the input, then the beam will be rendered from `[0,0,0]` to `[3,0,0]`. A node actor is also created for each global coordinate unless the user disables it.

```
Edof = np.array([
    [1,  2,  3,
     ↪  4,  5,  6,
     ↪   7,  8, 9,
     ↪  10, 11, 12]
    ])
```

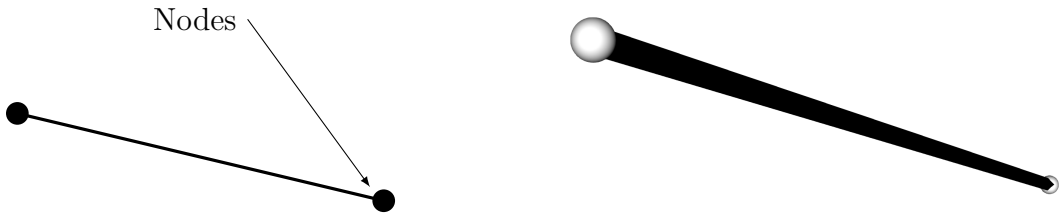
```
Dof = np.array([
    [1, 2, 3,
     ↪  4, 5, 6],
    [7, 8, 9,
     ↪  10, 11, 12]
    ])
```

```
Coord = np.array([
    [0, 0, 0],
    [3, 0, 0]
    ])
```

Listing 4.1: Edof, Dof & Coord example for a 3D-beam

4.1.1 Springs, bars & beams

These types of elements always have a single length along an axle, and can therefore be modeled using a shape as Vedo refers to it. For bars and beams, this is done using a `cylinder`, which simply draws a line from one point to another with a certain thickness, see Figure 4.1. For a spring element, there is a Vedo class `spring` used instead. This `spring` has the shape of a regular coil spring, but the user can also render springs using cylinders by changing an input variable if this is preferred.



(a) Theoretical element with nodes **(b)** Vedo visualization tool element

Figure 4.1: Spring/bar/beam element in CALFEM

4.1.2 Flow, solid & plates

For these types of elements, a different type of class is required. This is what Vedo calls a `mesh` (not to be confused with an FE mesh). This is a special actor, created by defining points in space, connecting them with lines, then defining which lines constitute a surface. These surfaces represent a volume. For flow/solid elements, these have one node in each corner of the rectangle, see Figure 4.2. For plates, there is only 4 nodes in between $-t/2$ & $t/2$.

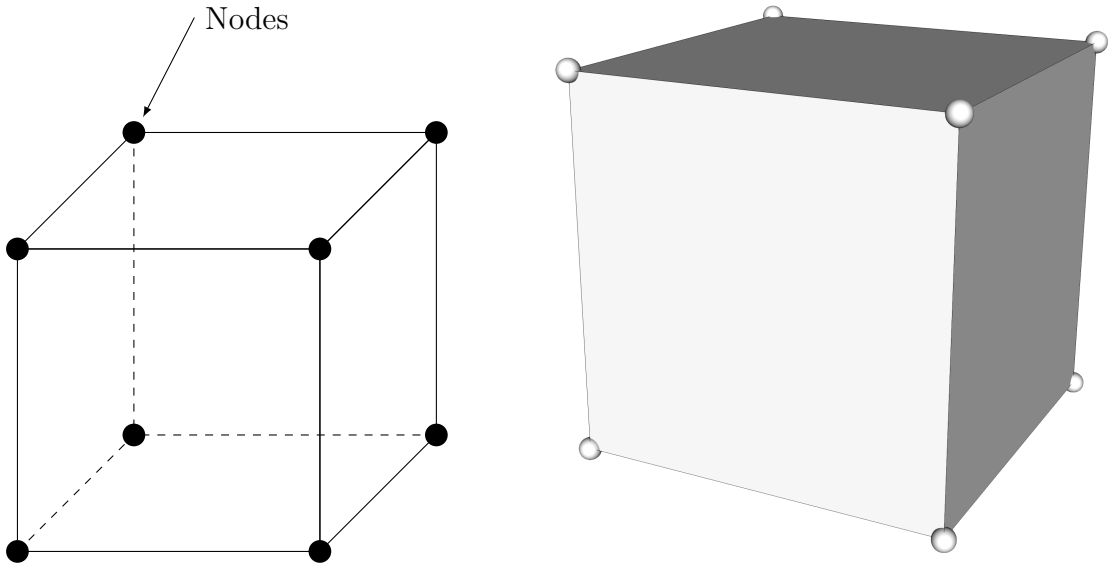
When drawing undeformed mesh, each element/node is its own mesh object, same as for elements in Chapter 4.1.1. For drawing displaced mesh however, this is implemented in a different way, see Chapter 4.1.3.

4.1.3 Deformed mesh

For the deformed mesh, a similar procedure to undeformed mesh (see Chapter 4.1) is done in order to find the global displacement for a node, then add it to the coordinates. If a displacement is given where the second node of the beam is displaced `-0.1` in the y-direction, the beam is rendered between `[0,0,0]` & `[3,-0.1,0]` instead. Rotational degrees of freedom are ignored for the purposes of visualizing elements, these are arbitrary.

A deformed mesh is handled the same as mesh for springs, bars, and beams (meaning each element is its own actor). For elements with volumes or surfaces however, a uniform mesh is created. This means that the whole mesh is one actor, with many points, lines, and surfaces. This is done for two purposes. First, using an unstructured grid for these elements enables better support for exporting to ParaView.

The other reason this is done is for color mapping (see Chapter 4.2). If each element



(a) CALFEM element

(b) Vedo element

Figure 4.2: 3D flow/solid element in CALFEM

is its own actor, unique points are created to represent its nodes. When elements are connected, this means there is a risk of discontinuities when mapping scalars as colors to the nodes. This came up as an issue during development. Essentially, if a mesh of solid elements was used for example, then each node would be eight nodes instead of one. This means this point could have eight colors mapped to it, which is undesired. Unstructured grids solve this by only allowing one point.

4.2 Color mapping for element & nodal values

In FE-modeling, results can be represented at nodes or different parts in an element. For some element types such as 3D flow/solids, internal element values are calculated at integration points, meaning each element has multiple values internally.

Vedo/VTK can map data at points and on surfaces [8]. Data values on surfaces will color the entire surface with a corresponding color. If there are multiple values for the stress in an element, for example, they either have to be interpolated to the nodes for them to be visualized continuously, or a single internal element value has to be calculated.

Due to the above-mentioned limitation in color mapping surfaces, visualizing element values requires one value per element. This single value, let's say stress, is then mapped to each surface representing that element. Element values can therefore be visualized by providing a column vector where each row contains the element scalar to be mapped to the surfaces of each element.

For visualizing nodal values, two methods are implemented, either a one-dimensional vector is provided, with as many rows as nodes. These are then applied globally. This could for example be a global displacement vector for a flow problem. If a

multidimensional array is provided, with rows as elements and nodal values as columns, a linear interpolation is done in order to map values to the global nodes.

There is no support for the input of scalar values at gauss points. If only two gauss points per dimension are used, there will be as many gauss points as nodes in the element (4 points in 2D, 8 points in 3D). This can be used as scalar input, but it will be interpreted as nodal values. The user can make an assume that the scalar values at the gauss points are approximate enough to be interpreted as nodal scalars.

The color mapping functionality is only implemented for displaced mesh, as the other mesh function is not intended for showing results. If the results from a model are to be visualized without displacements, the displacements can be omitted and the mesh will not be rendered as displaced.

4.3 Vectors & Principal stresses

Both vectors and principal stresses are supported using their separate functions. They function the same when rendering, they are rendered as cylinders in the middle of elements. Both support deformed elements if displacements are provided. See Figures 5.3a & 5.4d for examples.

4.4 Beam diagrams

Drawing section forces for beams has been implemented. Depending on what section force is plotted, dividing the beam into multiple segments (see Chapter 5.2) may be necessary. Normal and shear forces are supported, along with moment. The axes are inverted by default. The diagram's x-axis is always along the beam. Figure 4.3 shows an example of a moment diagram along a beam in the z-direction.

4.5 Animations

Animations have been implemented using linear interpolation between an undeformed and deformed state. Think of it as animating between the two types of meshes in a few steps. There are three different types of animations implemented, all using linear interpolation. The default is a simple animation from undeformed to deformed, then starting over again from the undeformed state.

The other ones essentially create a looping effect. The second option is to animate by including backward steps once the deformed state is reached. This means the first part of the animation goes from undeformed to deformed, then takes the same amount of steps from deformed back to undeformed.

The final type of animation is adding negative deformations to the loop. This is

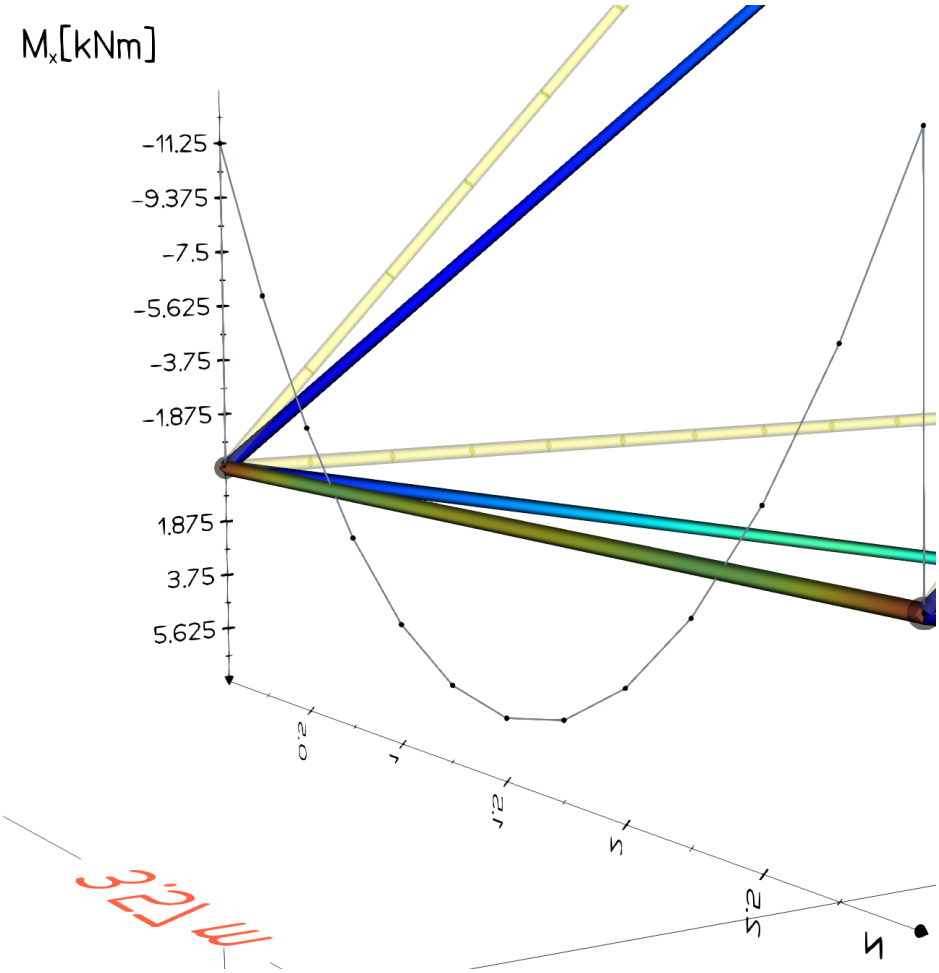


Figure 4.3: Moment diagram along 3D beam element divided into 8 segments

essentially what would be done for any dynamic model which is vibrating. When the undeformed state has been reached as in the second animation type, the animation repeats, but in the opposite direction.

The default amount of steps (between undeformed and deformed, for all types of animation) is 10. The user can choose the animation rate in milliseconds, i.e. how much time should pass between each frame. All 6 element types previously mentioned in this thesis are supported.

Animations can also be exported for use in ParaView, this is built into the animation function. When this is done, all steps are exported as individual files, which are then interpreted as an animation timeline by ParaView automatically.

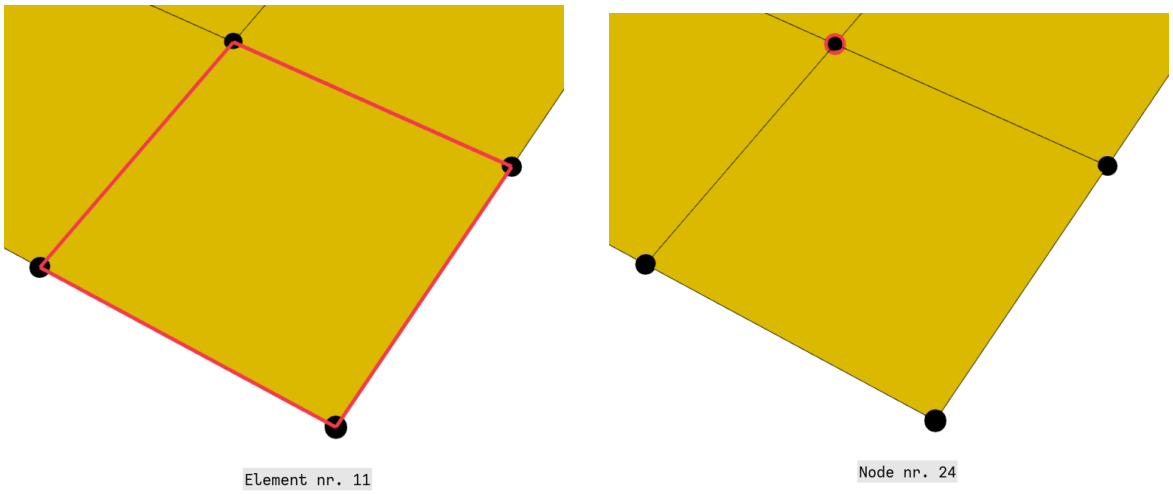
4.6 User interface & interaction

Initially, PyQt5 was intended to be used to implement a user interface for use with developed functions. Vedo allows for this, as it supports PyQt5. However, there were technical issues that couldn't be resolved easily. This issue was related to the mouse pointer and its position in the Vedo plotter. For some reason that has yet to be

resolved, the plotter interpreted the mouse pointer as being in a completely different place than it was, seemingly at random.

When interactivity was to be implemented, PyQt5 became an issue. It was instead chosen to implement interactivity using the Vedo method `addCallback`. This method adds an observer [7] which looks for an event. The event could be for example mouse movement, clicks, and keyboard input. By using a callback, clicking on elements and nodes to get their number has been implemented into the `draw_mesh` function.

Checking elements and nodes by number, along with highlighting them, has been implemented for undeformed mesh, see Figure 4.4. This is by default done by clicking elements/nodes, but figures can also be configured to highlight elements/nodes by hovering. This is in order to allow the user to check the mesh. If forces should be applied to a certain element/node, for example, the user can render the mesh and check it manually. This has not been implemented for deformed mesh, as it is not possible in Vedo unless each element is its own actor, see Chapter 4.1.3.



(a) Element numbering

(b) Node numbering

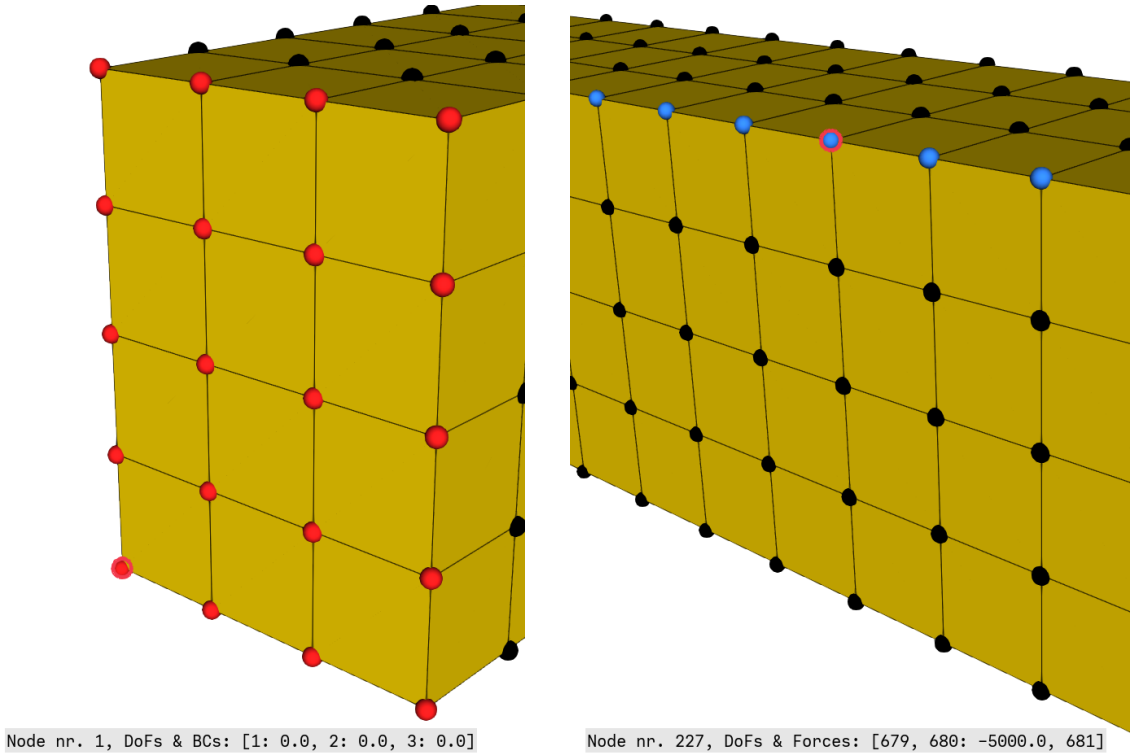
Figure 4.4: Highlighting & numbering by clicking/hovering

4.7 Forces & Boundary conditions

If the boundary conditions are given as input to an undeformed mesh, the nodes with a prescribed degree of freedom will be color-coded in red (instead of black, Figure 4.4). This color-coding only indicates that one or more degrees of freedom are prescribed, but not how many or which ones. In order to get this information, the nodes can be interacted with as in Chapter 4.6. This allows the user to see which degrees of freedom are prescribed, and what they're prescribed as, see Figure 4.5a.

This functionality is also supported for forces. These are colored in blue by default (colors for both boundary conditions & forces can be changed). Since forces can be applied to both nodes and elements, both can be color-coded this way. If one or

multiple forces are applied to an element, the user can get the directions of these forces in the same way as for nodes, i.e. by clicking. Figure 4.5b shows an example where a few nodes have an applied force in the negative y-direction. Note the text output at the bottom.



(a) Boundary conditions **(b)** Forces

Figure 4.5: Color coding nodes with boundary conditions & forces

4.8 Utilities

Various extra utility functions to supplement the visualized objects have been added. These include adding 2D text in the rendering window, projecting a mesh to a plane, scalar bars for color mapping, axes, etc. These are used in some examples in Chapter 5 and are described more in-depth in Appendix A. There are also some functions for getting coordinates based on degrees of freedom etc. These are used in the example presented in Chapter 5.4.

4.8.1 Keyboard shortcuts

Vedo supports so-called callbacks as mentioned in Chapter 4.6. The ones implemented are related to mouse clicks. Vedo also supports some built-in keyboard commands. Some of these are potentially useful for visualizations in CALFEM, while others are less useful. These standard commands can be disabled as a whole, but not one by one. Because some of them are useful, they have not been disabled. A full list of these commands will be printed to the command line if `h` is pressed.

4.9 Rendering

So far, visualizing elements has been referred to as visualization. In order to visualize elements in a window, Vedos `Plotter` class is required, see Chapter 3.3. In this Chapter, the implementation of this class is presented. Rendering is part of what the `Plotter` class handles. Multiple plotters can be used in order to have multiple windows.

For starting and handling plotters, a class `VedoMainWindow` was created. This class implements methods for interacting with the visualization and is required to have an instance. Rendering is not started upon initialization of this class, but it contains the method for this, (see Listing 4.4) The class mainly handles attributes and interaction. In Listing 4.2, some variables used to store data are shown. When the user runs a function that creates data to be rendered, these are appended to one of the lists here.

```
def __init__(self):
    self.fig = 0 # Figures
    self.meshes = [[]] # Meshes/elements
    self.nodes = [[]] # Nodes
    self.msg = [[]] # Optional text messages
    self.proj = [[]] # Optional projections
    self.rulers = [[]] # Optional rulers

    ...
```

Listing 4.2: Initializing the `VedoMainWindow`-class.

This main class keeps track of an integer called `fig`, for figure. This variable allows for the use of figures in a similar way to MATLAB. If the user only needs a single figure for visualizing, using the visualization functions works straight away. If a user wants to plot things separately however, the method in Listing 4.3 is used. This allows for using `figure(2)` to render the output from the following functions in figure 2, for example.

```
def figure(fig):
    ...

    if fig < 1:
        print("Please give a positive integer (> 0)")
        sys.exit()
    else:
        plot_window.fig = fig - 1

    ...
```

Listing 4.3: Figure method for handling multiple Vedo plotters

The method created for rendering all figures is shown in Listing 4.4. This method loops through the created figures and renders all actors associated with it. These actors are created by the user indirectly by calling different functions, before this point. For instance, each row of `self.meshes` is a list where each row contains all the Vedo mesh-objects for the specific figure. If mesh objects have been created by `draw_mesh` or `draw_displaced_mesh`, these are rendered.

```
def render(self):
    for i in range(self.fig+1): # Loops through figures
        opts = dict(axes=4, interactive=False, new=True,
        ↪  bg='k', title=f'Figure {i+1} - CALFEM vedo
        ↪  visualization tool') # Options for plotter
        plt = v.show(self.meshes[i], self.nodes[i],
        ↪  self.click_msg, **opts) # Plotter
        plt.addCallback('mouse click', self.click) #
        ↪  Callback for mouse click

        for j in range(len(self.msg[i])):
            plt.add(self.msg[i][j]) # Add text

        for j in range(len(self.proj[i])):
            plt.add(self.proj[i][j]) # Add projections

        for j in range(len(self.rulers[i])):
            plt.add(self.rulers[i][j]) # Add rulers

    v.interactive() # Shows interactive visualization
```

Listing 4.4: `render`-method in the `VedoMainWindow`-class.

4.9.1 Instantiation of Vedo classes

The user starts the Vedo visualization by running the function `show_and_wait`, shown in Listing 4.5, at the end of their file. This runs the rendering in Listing 4.4. The rendering is started through the use of a class `VedoPlotWindow` which handles class instances [25]. This class was created to make sure there is only one instance of the `VedoMainWindow` class. This class is shown in Listing 4.6, it checks if there is an instance of the `VedoMainWindow` class. If one exists it returns it. If no instance exists, it creates one. This allows for very simple interfacing with the `VedoMainWindow` instance. This is useful for implementing simple functions like the ones in Chapter 4.8. If a class, like `VedoMainWindow`, is called directly, by default a new instance is created [26]. In order for the visualization to work correctly, the user must therefore always use `show_and_wait`.

```
def show_and_wait():
    app = init_app()
    plot_window = VedoPlotWindow.instance().plot_window
    plot_window.render()
```

Listing 4.5: `show_and_wait` function used for starting the Vedo visualization

```
def init_app():
    global vedo_app
    vedo_app = VedoPlotWindow.instance()
    if vedo_app is None: vedo_app = VedoMainWindow()
    return vedo_app

class VedoPlotWindow:
    __instance = None
    @staticmethod
    def instance():
        """ Static access method. """
        if VedoPlotWindow.__instance == None:
        ↪  VedoPlotWindow()
        return VedoPlotWindow.__instance

    def __init__(self):
        """ Virtually private constructor. """
        if VedoPlotWindow.__instance is None:
            VedoPlotWindow.__instance = self
            self.plot_window = VedoMainWindow()
```

Listing 4.6: Starting the Vedo visualization in the `VedoMainWindow`-class.

4.10 Error handling

Error handling for functions is implemented in a function called `check_input`. This function ensures user input is correct in terms of the `Edof`, `Coord` and `Dof` matrices. If the user applies a deformation, element or nodal values, these are also checked. For example, if a user gives `Coord`- and `Dof`-marticies with different amount of rows, the function will stop and return the following:

```
Beam element: Number of rows in Coord & Dof does not
 ↪  correspond, please check them along with number of nodes
```

This function is used by the functions for drawing meshes. Since the element types affect the sizes of these matrices, they are checked following the element type the user

gave as input. If the user gives the wrong element type, the function cannot check it correctly. This unfortunately also means that an error message can be returned, indicating the wrong error. It is therefore important to make sure that the correct element type is assigned by the user. To aid in this, the element type is also included in the error message for all error messages of this kind, as above. If an element type outside the 1-6 range is given, an error message will be returned listing the different types. Error handling has not been implemented for various miscellaneous functions such as adding text, scalar bars, etc. The focus has been on the more complicated functions, as troubleshooting these would be very difficult for the user otherwise. This is not a substantial issue for the simpler functions. If error handling is needed for these, checking the manual/source is simple. Comments describing the input/output for each function (intended to be called by a user) are also in the source code.

4.11 General issues during development

In the initial stages of development, cylinders were not used to represent bar and beam elements. These were represented with simple line-actors instead. Both VTK and Vedo allow for rendering these as cylinders. This would have simplified meshes for these elements. However, it was discovered that the rendered cylinders don't behave as intended. They behaved as lines, which in VTK means that they remain the same scale regardless of zoom level. This had the effect of rendering the beams/bars as tiny upfront, and massively thick when zooming out. The creator of Vedo, Marco Musy was contacted via Github for recommendations on proceeding. He explained that the lines are still lines (see source object in Chapter 3.1), meaning changing their behavior is not possible. This correspondence lead to the current implementation of these element types. If rendering the lines as cylinders would have worked, an implementation using a mesh object (in VTK), as for the other elements, would have worked. In this case, the mesh would lack surfaces in modeling, only containing points and lines.

Vedo was first released in 2019 and is therefore fairly new in terms of libraries. Some stability issues can be expected. There seem to be some issues with the Vedo plotter class. The class works inconsistently depending on the use case, even when used according to the manual. The implementation of this class has also changed a few times, as new issues were encountered. One issue that couldn't be resolved, involved multiple figures on Linux, where only one figure could be interacted with. Once again the creator was contacted for resolving this. He gave recommendations for VTK versions to use, these are given in Appendix A.

5 Usage examples

This Chapter intends to show the variety of functionality implemented in this module using a few examples. These examples have to a large extent been used during development and testing. These examples are presented more extensively in Appendix A, an overview is given in this Chapter.

The focus when creating these examples has been to illustrate what the visualization module can do. These are not to be taken as guidance as to how actual FE analyses of these problems should be done. For this reason, no verification that the FE modeling is reasonable (such as convergence) has been made.

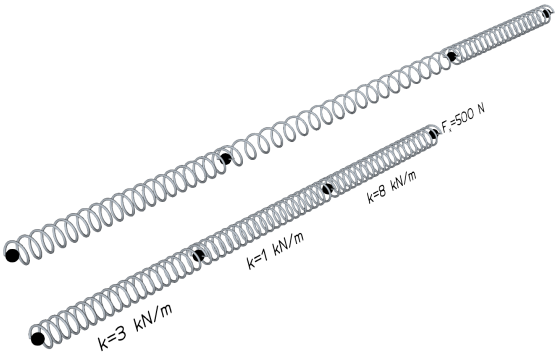
5.1 Simple spring model in 3D

This example implements a 1D spring but renders it in 3D-dimensional space. In addition, it serves to show how offset works. The undeformed mesh is rendered, then the displaced mesh is added but offset above the other mesh. This allows for visualizing the displacements after solving, in the same figure. See Figure 5.1a. Some text is added to the plotter to illustrate the stiffnesses of the different springs and the force applied.

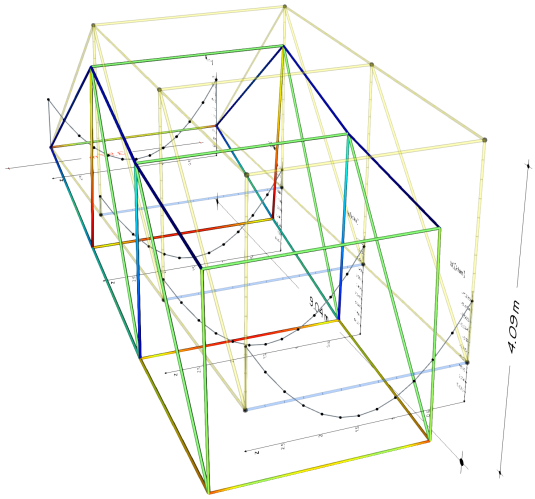
5.2 3D truss model using symmetry

This is an example of implementing both 3D beam and bar elements. It shows how to handle multiple meshes in the same figure using transparency, also text output features in the corners of the plotter window. It demonstrates how different element types can be visualized in the same figure.

The transverse beams at the bottom are loaded with a uniform load, which gives a non-linear moment distribution along these. Therefore, the beams are divided into 8 segments (not separate elements). The CALFEM function `beam3s` then allows for calculating the section forces in these segments. The `draw_displaced_mesh` function has been developed with this function in mind for beams, allowing for visualization of varying section forces/stresses in beam segments. The beam is also loaded with a point load sideways at the top. The resulting least favorable normal stresses are color mapped to all elements. Measurements are drawn then finally, moment diagrams are drawn along the transverse beams, see Figure 5.1b.



(a) Spring model with undeformed & deformed meshes in the same figure using offset

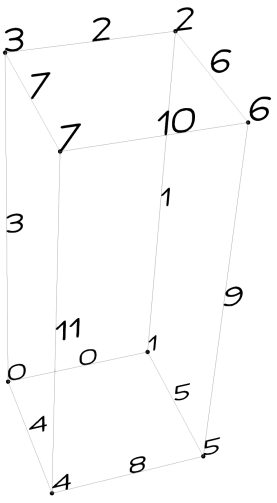


(b) Truss model using bars & beams and color mapping least favorable normal stress

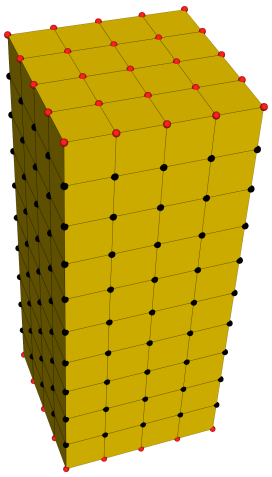
Figure 5.1: Spring, bar & beam examples

5.3 3D heat flow model

This example illustrates how the functions can be used alongside the geometry & meshing modules in CALFEM for Python. First, a geometry is defined using the built-in geometry module in the Python version, then it is visualized (Figure 5.2a). A mesh is then created using the meshing module and is visualized for verification (Figure 5.2b). Boundary conditions are prescribed using the same module, two arbitrary elements are prescribed as heat sources. Then an FE analysis is done to visualize the heat flux using vectors and color mapping (Figure 5.3a) along with temperature distribution (Figure 5.3b). Nodes with prescribed degrees of freedom are shown in red in Figure 5.2b. Figures 5.3a & 5.3b also show use two different implementations of adding axes to a visualization.

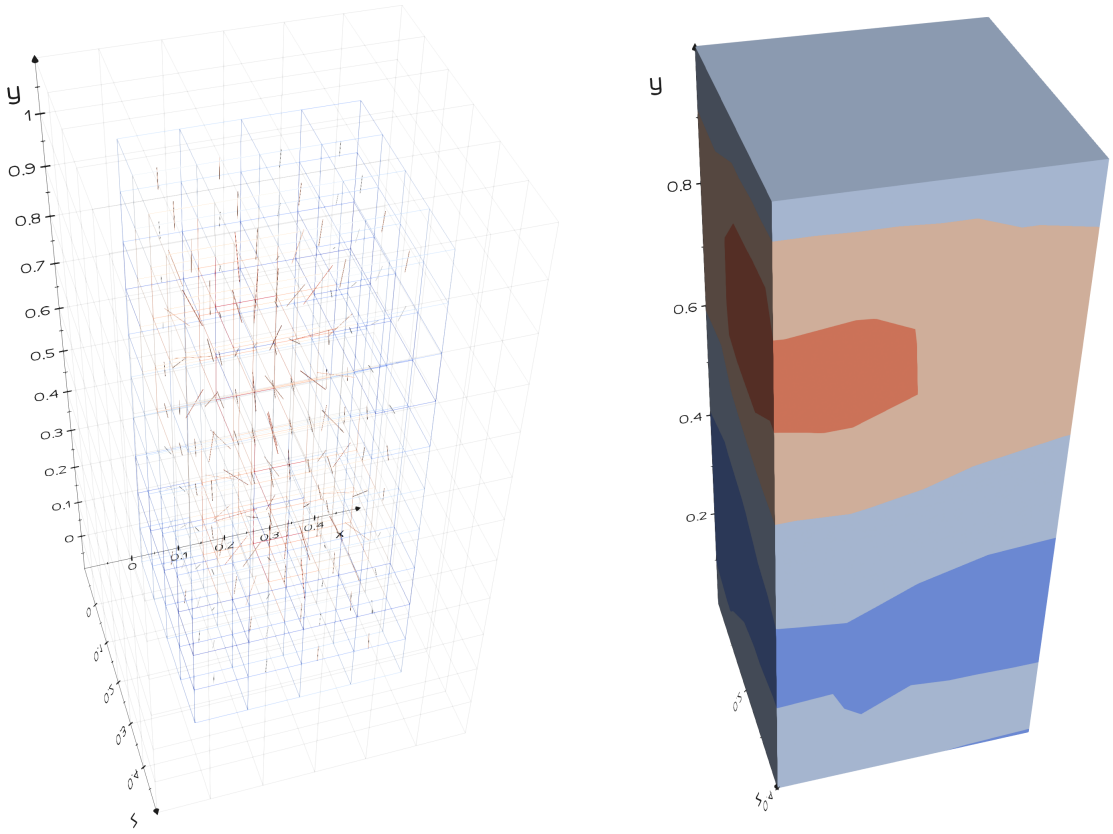


(a) Geometry



(b) Mesh

Figure 5.2: 3D flow example in CALFEM



(a) Heat flux vectors

(b) Temperature distribution

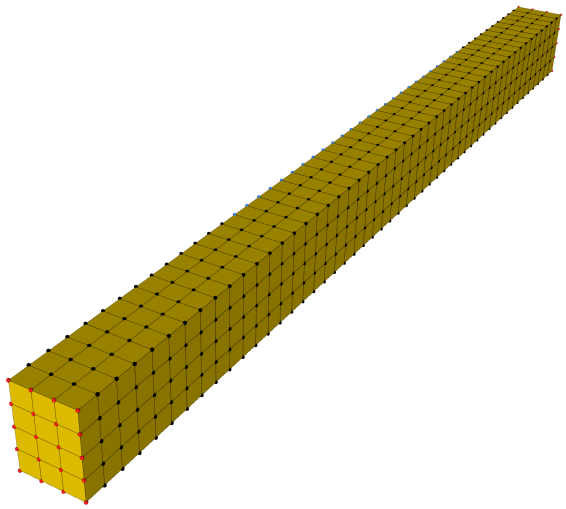
Figure 5.3: Results for 3D flow example in CALFEM

5.4 3D solid model using import & export

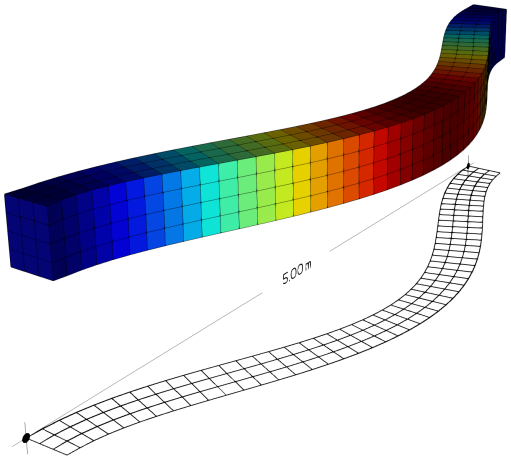
This is a more extensive example illustrating how a rectangular beam can be modeled using solid elements. Multiple figures are used to visualize different aspects of the beam model. The example mostly uses MATLAB code which is imported. One of the rendered meshes is then exported to VTK-format for use with ParaView, exemplifying a potential workflow from MATLAB to ParaView.

In the example, two different analyses are performed. One linear static model which is loaded with uniform self weight and eccentric point loads (Figures 5.4c & 5.4d), and one modal analysis (Figure 5.4b). Figures 5.4b & 5.4c uses deformation scalefactors. For the linear analysis, two gauss points per dimension are used, eight in total.

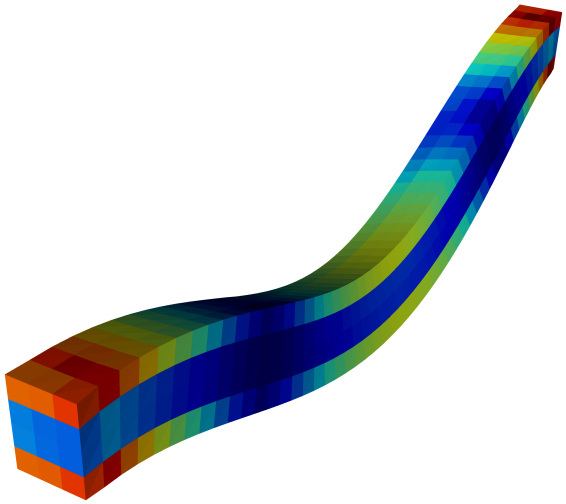
The two analyses are done in MATLAB, then imported for visualization only. Calculations of von Mises stresses from the static analysis and displacements from the modal analysis are converted to element scalars. The latter is done using functions from Chapter 4.8, showing how a user can utilize them as well. Finally, the undeformed mesh (along with nodes) is visualized (Figure 5.4a). Here, red nodes indicate nodes with boundary conditions set to zero, and blue nodes indicate nodes where a point load is applied. The modal analysis in Figure 5.4b also includes a 2D projection of the deformation on the xz-plane. The von Mises stresses from the static analysis are visualized using element scalars in Figure 5.4c.



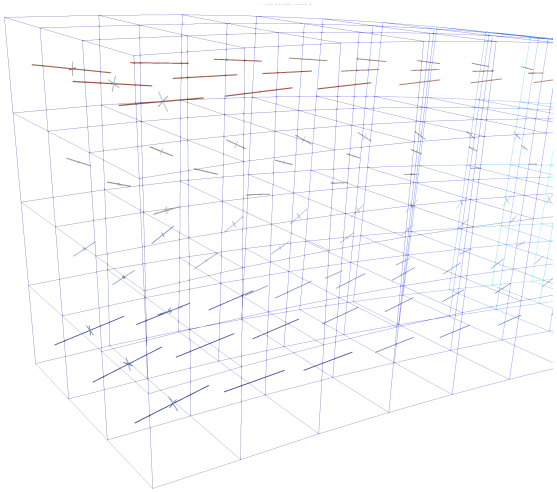
(a) Undeformed mesh with color coded boundary conditions & forces at nodes for static analysis



(b) Deformed mesh showing the eigenvector for the lowest eigenmode using deformation, color mapping & a projection



(c) Deformed mesh from static analysis with color mapped von Mises stresses at elements



(d) Deformed mesh using wireframe mode with principal stresses from static analysis at the center of each element

Figure 5.4: 3D solid example in CALFEM

Principal stresses are also calculated, then visualized as vectors in Figure 5.4d. This visualization is similar to Figure 5.3a, however with three vectors instead, representing the principal stresses. Figure 5.4d uses wireframe mode and shows the principal stresses at one edge of the beam. Note that the stresses are perpendicular to each other, as would be expected.

This model has also been used as a basis for the implementation of animations. Figure 5.4b is animated as a vibration, meaning the negative deformation (deformed in the other direction) is included. This is done in Vedo, and the result is exported to ParaView and animated. The resulting visualization in ParaView can be seen in Figure 5.5.

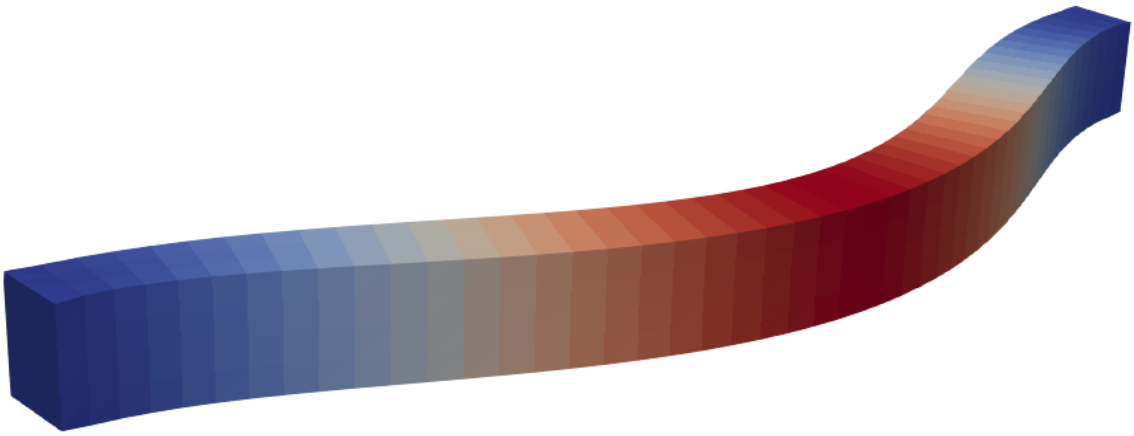
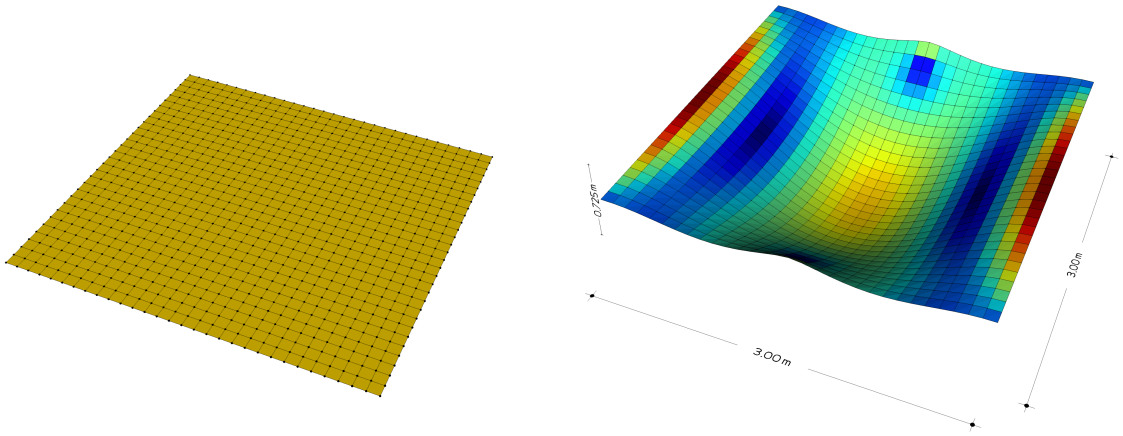


Figure 5.5: Modal analysis results from Figure 5.4b animated in ParaView

5.5 Plate model in 3D

Visualizing a plate in 3D allows for visualizing displacements in the third dimension, and at the same time visualizing moments/stresses as a colormap applied to the elements or nodes. This is not possible in 2D, where the displacements would need to be shown as a colormap.

In the example, a 3x3 m plate mesh is created, then visualized (Figure 5.6a). After checking the mesh, boundary conditions and forces are applied and the problem is solved. Two opposite edges of the plate are fixed. Along the other two edges, the mid nodes degree of freedom in the z-direction is set to zero. This could be the effect of support from columns here for example. After solving, the deformed mesh is visualized and von Mises stresses are color mapped to elements, resulting in Figure 5.6b. A scale factor is applied to the deformation and rulers are added to get the dimensions and deformation.



(a) Undeformed mesh **(b)** Deformed mesh & max. von Mises stresses

Figure 5.6: 2D plate example in CALFEM

6 Discussion

The survey of visualization needs unfortunately had few responses. The study was conducted as a survey as opposed to an interview study to try and get as many responses as possible. Conducting interviews would likely have given even fewer responses and not much more information. The intention was to get relatively simple feedback from many staff members, not in-depth input from a few. The study was still useful as it gave input from two departments and some clear needs could still be identified. The results were used as input on what elements and types of visualization to focus on.

Initially, after the study of visualization libraries in Python had been conducted, VTK was chosen for development. It was at that point motivated by its extensive use and reliability, vast amounts of documentation, and compatibility with ParaView. However, it became apparent that for this project, the workflow in VTK was not optimal. This was due to the extensive amount of classes needed. As many types of visualizations for different elements were needed, the functions would become very complicated. Using VTK and adopting it for CALFEM, would risk developing something Vedo-like in the process, at least for the functionality needed. The documentation for VTK was sometimes of little use as it is intended for use in C++.

At that point, the study was expanded and Vedo along with PyVista was evaluated. This turned out to be a good decision. After evaluating Vedo and PyVista, the functionality developed using VTK up to that point was able to be adopted for these libraries with very little effort. This in itself proved that they were much more suitable for use in the development. After studying both libraries, it is difficult to differentiate them in many ways. This is why they are listed together in Chapter 2.4.3. In the end, Vedo was used as it seemed to have a slight edge in ease of use. Either one could have been used in the project.

The reason for using Qt, in the beginning, is that the visualization functionality was conceptualized more as a complete tool with menus. As this was deemed to have few benefits, compared to the amount of work it would take to implement, this was disregarded in favor of adding additional functions in CALFEM for Python. Some of the functionality would have been interaction with the model (like changing representation of global axis during rendering, hiding objects, etc). Other things were import/export. Some functionality that was initially supposed to use menus, turned out to be easier to implement and even work better by using Vedo classes directly, as querying element/node numbers. Qt also caused problems with interaction in Vedo (see Chapter 4.6). The cause could not be determined, it could be due to an issue in Vedo. This had little effect on the outcome of the project, as most of the functionality mentioned above was implemented using Vedo instead.

Animations were not part of the initial goals, however seeing there was a demand for it from staff, it was nevertheless implemented. The resulting animation functionality is limited (see 4.5). Vedo provides several examples of how animation can be

implemented, none of which provide a good way to exit animations. To visualize a continuous animation, a keyboard press has to be used. This implementation was used for CALFEM, the issue with it is it is not consistent with how other figures are closed. Thanks to the implemented export functions, animations can be exported directly to ParaView. This is arguably a better solution, as ParaView allows the user to do much more with animations. There is however a value to being able to visualize it immediately in CALFEM. The implemented animation function in CALFEM should be considered experimental, and exporting animations to ParaView is recommended.

Visualization functions for most steps in the FE workflow have been developed for CALFEM. Geometry, mesh, and many results are supported (see Figure 1.1). Visualizing forces and boundary conditions are supported in a limited capacity. If vectors with degrees of freedom are given to the `draw_mesh` function, the nodes containing the degrees of freedom in this list are color-coded. To get the boundary condition/force applied in the node, these can also be provided. This is a very simple implementation, visualizing a force with a vector in a direction is not possible.

7 Concluding remarks

A recurring theme of this thesis is the focus on usage in teaching the Finite Element Method. This philosophy of CALFEM has been a clear guideline. The developed module and functions are meant to be a teaching tool, expanding CALFEM. As most of the functionality is implemented as functions, documentation is a very important aspect for it to be successfully used in a learning situation. Functions are documented both in source code as well as online documentation provided with CALFEM for Python, in addition to the manual in Appendix A.

7.1 Studies

The initial goal of this project was to conduct three studies to aid in the development moving forward. These studies, consisting of analyzing the current visualization functionality in CALFEM for Python, needs for further visualization functions, and libraries to use, were conducted successfully. The study of current visualization functionality helped in terms of development as it gave an increased understanding of the intended functionality and use of functions in CALFEM for Python. The study of needs for visualization tools had few responses. However, this study can be considered supplementary, as some needs were already known. One of the identified needs besides the survey where more 3D functionality, which has now been expanded.

The study of suitable Python visualization libraries was successful in evaluating a suitable library for use in CALFEM for Python. Many options were studied and evaluated for ease of use, performance, and stability. A very functional, simple-to-use library, Vedo, was chosen for implementing the required functionality. After working with it, it seems to be better maintained than initially thought. It is, in my view, an excellent basis for expanding visualization in CALFEM further.

To summarize what has been studied:

- ✓ Existing visualization tools in CALFEM for Python
- ✓ Needs for visualization
- ✓ Python libraries suitable for implementing visualization

7.2 Visualization

Regarding the actual visualization in CALFEM, much has been added in terms of functionality in 3D. All element types from the MATLAB version can be visualized in some way. Support for some elements is more limited, while flow, beam, bar, and solid elements have extensive functionality. Export to VTK has been implemented, allowing for visualization in the application ParaView. In addition, importing from Matlab has been implemented. Both import and export are simple, single-line functions.

Some things beyond the initial scope of the project have also been implemented. During the second study on needed functionality, animations turned out to be a desired feature. An effort was made to implement some sort of functionality for this, however not extensively. Animations support all elements the other functions support. Three different types of (linear) animations are supported, see Chapter 4.5.

To summarize what has been implemented:

- ✓ 6 types of elements (spring, bar, flow, solid, beam & plate)
- ✓ Geometry, undeformed mesh & deformed mesh
- ✓ Color mapping by scalars for nodes & elements
- ✓ Vectors
- ✓ Principal stresses
- ✓ Section force diagrams for beams
- ✓ Animations for all 6 element types
- ✓ Import from MATLAB & export to ParaView
- ✓ Error handling
- ✓ 2D mode
- ✓ Other utilities

8 Future Work

Throughout this work, a user interface built on PyQt5 was at times used to some extent. In the end, it was decided to not implement this. The foremost reason was that it interfered with the functionality related to the mouse pointer in Vedo. This meant getting element/node information would be impossible if Qt was used. This was considered too important to disregard for a real-world use case. Going forward, this could be looked into and solved. If done, it would somewhat improve usability and reduce the need for some scripting in Python.

Animations can be greatly expanded upon. The current implementation is fairly crude, only using two different states of the mesh and linearly interpolating steps in different ways. The Finite Element Method has many possibilities of dynamic, time-dependent, and non-linear modeling. A few methods for this exists in the current versions of CALFEM for both Python and MATLAB. Adding dedicated methods for this would make this tool more complete for teaching, also for more advanced FE modeling. For example, manually adding keyframes to the animation timeline could be added.

Vedo allows for interaction with the PyPlot library. This allows for plotting 2D graphs in the 3D space of the Vedo plotter. This is not something that has been explored in this thesis, however, some examples provided by Vedo show this possibility [14]. This could then be used for more extensive plotting of graphs in the same figure as the model.

Isolines have not been implemented in any form. Based on the survey of visualization needs, (Chapter 2.2.2) it was deemed to be little demand for it for all element types. If the tools are to be expanded for certain types of problems however, Vedo supports implementing them.

Bibliography

[1] Per-Erik Austrell, Ola Dahlblom, Jonas Lindemann, Anders Olsson, Karl-Gunnar Olsson, Kent Persson, Hans Petersson, Matti Ristinmaa, Göran Sandberg and Per-Anders Wernberg. *CALFEM - A Finite Element Toolbox version 3.4*. KFS i Lund AB, 2004.

[2] Jonas Lindemann. *CALFEM for Python Documentation*. 2021. URL: https://calfem-for-python.readthedocs.io/_/downloads/en/latest/pdf/.

[3] Christoph Schäfer. *Quickstart Python. An Introduction to Programming for STEM Students*. Tübingen, Germany: Springer, 2021.

[4] Jonas Lindemann. *Ingenjörens guide till Python*. 2019.

[5] Philipp Rudiger, James A. Bednar, Julia Signell and Jean-Luc Stevens. *Python tools for data visualization*. 2019. URL: https://pyviz.org.

[6] Daniel Weiskopf, Kwan-Liu Ma, Jarke J. van Wijk and Helwig Hauser. "SciVis , InfoVis – bridging the community divide ? !" In: 2006.

[7] William Schroeder, K. Martin and William Lorensen. *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics*. Jan. 2006.

[8] Andrew C. Bauer, Berk Geveci and Will Schroeder. *ParaView Catalyst User's Guide*. 2019. URL: https://www.paraview.org/paraview-downloads/download.php?submit=Download&version=v5.9&type=data&os=Sources&downloadFile=ParaViewCatalystGuide-5.9.1.pdf.

[9] W.J. Schroeder, K.M. Martin and W.E. Lorensen. "The design and implementation of an object-oriented toolkit for 3D graphics and visualization". In: *Proceedings of Seventh Annual IEEE Visualization '96*. 1996, pp. 93–100. DOI: 10.1109/VISUAL.1996.567752.

[10] Berk Geveci, Will Schroeder, A Brown and G Wilson. "VTK". In: *The Architecture of Open Source Applications* 1 (2012), pp. 387–402.

[11] Patrick O'Leary, Sankhesh Jhaveri, Aashish Chaudhary, William Sherman, Ken Martin, David Lonie, Eric Whiting, James Money and Sandy McKenzie. "Enhancements to VTK enabling scientific visualization in immersive environments". In: *2017 IEEE Virtual Reality (VR)*. 2017, pp. 186–194. DOI: 10.1109/VR.2017.7892246.

[12] Prabhu Ramachandran. *Mayavi documentation*. 2021. URL: https://mayavi.readthedocs.io/en/latest/installation.html.

[13] Prabhu Ramachandran and Gaël Varoquaux. "Mayavi: a package for 3D visualization of scientific data." In: (2010). URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=edsarx&AN=edsarx.1010.4891&site=eds-live&scope=site.

[14] Marco Musy et al. “Vedo, a python module for scientific analysis and visualization of 3D objects and point clouds”. In: (2021). DOI: 10.5281/zenodo.4287635. URL: https://doi.org/10.5281/zenodo.4287635.

[15] C. Bane Sullivan and Alexander Kaszynski. “PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)”. In: *Journal of Open Source Software* 4.37 (2019), p. 1450. DOI: 10.21105/joss.01450. URL: https://doi.org/10.21105/joss.01450.

[16] Anders Logg, Kent-Andre Mardal, Garth N. Wells et al. *Automated Solution of Differential Equations by the Finite Element Method*. Ed. by Anders Logg, Kent-Andre Mardal and Garth N. Wells. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: 10.1007/978-3-642-23099-8.

[17] Nicholas Sharp et al. *Polyscope*. 2019. URL: www.polyscope.run.

[18] L. Stemkoski and Pascale M. *Developing Graphics Frameworks with Python and OpenGL*. 1st edition. CRC Press, 2021. DOI: 10.1201/9781003181378. URL: https://doi.org/10.1201/9781003181378.

[19] Johan Nysjö. *Introduction to Python and VTK*. 2014. URL: https://www.cb.uu.se/~aht/Vis2014/lecture2.pdf.

[20] William J Schroeder, Lisa Sobierajski Avila and William Hoffman. “Visualizing with VTK: a tutorial”. In: *IEEE Computer graphics and applications* 20.5 (2000), pp. 20–27.

[21] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke and Travis E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[22] Andreas Ottosson. *Implementering av CALFEM för Python*. swe. Student Paper. 2010.

[23] Andreas Edholm. *Meshing and Visualisation Routines in the Python Version of CALFEM*. eng. Student Paper. 2013.

[24] Karl Eriksson. *CALFEM Geometry Editor - Implementing an interactive geometry editor for CALFEM*. eng. Student Paper. 2021.

[25] Mark Lutz. *Learning Python : Powerful Object-Oriented Programming*. Sebastopol, United States: O’Reilly Media, Incorporated, 2013. ISBN: 9781449355715. URL: http://ebookcentral.proquest.com/lib/lund/detail.action?docID=1224732.

[26] Hans Petter Langtangen. *A Primer on Scientific Programming with Python*. 4th. Springer Publishing Company, Incorporated, 2014. ISBN: 3642549586.

Appendix A: Manual

Vedo visualization in CALFEM

This Appendix is only for the Vedo visualization tools in CALFEM for Python. For full CALFEM for Python documentation, please see:

`https://calfem-for-python.readthedocs.io/en/latest`

CALFEM for Python source code and examples in this Appendix can be found at:

`https://github.com/CALFEM/calfem-python`

For viewing exported models, see ParaView documentation at:

`https://docs.paraview.org/en/latest`

Contents

A.1 Installation & requirements

Some form of Python installation is required, a scientific Python distribution like Anaconda can be used, or a regular installation of Python. Python 3.8 or later is recommended as the tools have been tested on these versions. Vedo version 2022.0.1 or later is required for all features and VTK version 9.1 or later is recommended. Issues with interaction using multiple figures can occur on VTK versions 9.0.X.

Using the visualization tools requires Vedo and the Python version of CALFEM. These can be installed in Anaconda by running the following in the IPython terminal:

```
conda install vedo
conda install calfem
```

For installing natively using Python, the following can be run in the terminal of your operating system:

```
pip install vedo
pip install calfem
```

The module can be imported by writing `import calfem.vedo` in your Python file. To avoid potential conflicts with other packages, the package can be imported with a reference as well. This is done by writing `import calfem.vedo as cfv`.

A.2 Basic visualization

The main visualization functions are dependent on the `Edof`, `Dof` & `Coord` matrices widely used in CALFEM. Each row in `Dof` describes the degrees of freedom for each of the nodes, the rows in `Edof` assign these degrees of freedom to the elements. `Coord` describes the coordinates for the nodes. The functions use the Python version of the `Edof` matrix, which means there is no indication for the element number in the first column, instead the row number denotes the element number. In addition to this, specifying the element type is required when visualizing any mesh. The element types are given below in Table A.1.

Type	Element	Ex. function in CALFEM
1.	Spring	`spring1e`
2.	Bar	`bar2e`
3.	Flow	`flw3i8e`
4.	Solid	`soli8e`
5.	Beam	`beam3e`
6.	Plate	`platre`

Table A.1: Element types

In order to visualize, one or more of the main functions `draw_geometry`, `draw_mesh`, `draw_displaced_mesh` or `animate` must be run first. After any of these functions, the visualization is run using `show_and_wait`. The user can run as many main functions as they want, before showing the renderings using `show_and_wait`. If this is done using `figure`, each visualization will open in separate windows. Essentially, running functions adds objects (actors, text etc) to the figure currently selected (1 by default), and the objects are then rendered using `show_and_wait`.

The mesh functions `draw_mesh` and `draw_displaced_mesh` have many optional input parameters. In the following example, only basic functionality is shown, with some optional parameters. For complete explanations of all functions and input parameters, see the function reference A.7. Section A.5 contains more extensive and complete examples.

Let's go through a simple example where a **geometry**, **undeformed mesh**, and **deformed mesh** is visualized. The geometry is created using the CALFEM for Python module for it. Points, lines & surfaces are visualized. Points are required, the others are optional. If surfaces are included, lines are also required. This is done as follows:

```
g = cfg.Geometry()
⋮
points = g.points
lines = g.curves
surfaces = g.surfaces
cfv.draw_geometry(points,lines,surfaces)
```

When meshing is done, the user has the `Edof`, `Coord` and `Dof` matrices available. Let's use solid elements, (element type 4, see Table A.1) now the user can visualize the undeformed mesh in a separate figure:

```
cfv.figure(2)
cfv.draw_mesh(edof,coord,dof,4)
```

After the FE problem is solved, the results can be visualized in a third figure. Let's say the displacement (`a`) is quite small and needs to be increased to see the deformed shape properly. An optional scale factor of 10 is therefore applied and noted using text in the top-left corner. If von Mises (`vM`) stresses have been calculated in each element, these can be visualized as scalars using a colormap and adding a scalar bar. If a scalar bar is used, it must be added after `draw_displaced_mesh`. Finally, all created figures are shown. This is done as follows:

```
cfv.figure(3) # Omit this to use same figure as undef. mesh
dS = 10
cfv.add_text(f'Def. scale: {dS}',pos='top-left')
cfv.draw_displaced_mesh(edof,coord,dof,4,a,vM,def_scale=dS)
cfv.add_scalar_bar('von Mises stress [MPa]')
cfv.show_and_wait() # Starts visualization
```

A.3 Animations

Animation is done by linear interpolation in a few steps between an undeformed and a deformed state. The number of steps can be chosen, as well as the time difference between each step. The default is 10 steps (excl. undeformed mesh) with 50ms in between. The results can be exported to ParaView. Animations start over once finished, and has to be exited by pressing `ESC`.

Three animation modes are supported. By default, the animation starts at the undeformed state (0) and animates steps to the deformed state (1). The loop animation mode also includes backward steps from the deformed to the undeformed state. The negative mode does what the loop mode does, but when the undeformed state is reached, the animation is reversed to the negative deformed state (-1) instead. This is essentially representing a vibration. See below:

$$\begin{aligned} \texttt{Default}: &\ 0 \rightarrow 1 \\ \texttt{Loop}: &\ 0 \rightarrow 1 \rightarrow 0 \\ \texttt{Negative}: &\ 0 \rightarrow 1 \rightarrow 0 \rightarrow (-1) \rightarrow 0 \end{aligned}$$

Note that the number of steps used as input for the animation function denotes how many steps are used between each state (incl. deformed). This means that the loop mode has approximately double the total amount of steps as the default mode. The same applies when going from loop to negative mode.

Let's continue the example from Section A.2. The last step (results) can be animated. The number of steps is given after the deformation. Let's use 8 steps with 100ms in between. A loop is used and the animation is exported for use in ParaView. This can be done as follows:

```
cfv.figure(4) # Animations require separate figure
cfv.animate(edof,coord,dof,4,a,vM,8,dt=100,loop=True,def_sca⌋
 ↪  le=dS,export=True,file='anim/exv4')
cfv.show_and_wait() # Starts animation
```

A.4 Import & export

A.4.1 Import from Matlab

Before any CALFEM data can be imported, it needs to be saved to a `.mat`-file in Matlab. This is be done as follows:

```
save('exv4.mat','Coord','Dof','Edof')
```

For importing Matlab data, `import_mat` is used. It allows for importing .mat-files in a way that is suited to CALFEM. Its suggested use is:

```
Edof,Coord,Dof = import_mat('file', ['Edof','Coord','Dof'])
```

It takes one required argument, the file name to be imported (with or without .mat-extension). It also takes an optional argument in the form of a list. The output will be ordered the same as the list. This ensures the data is imported correctly, omitting this list risks swapping around the variable names (for example `Edof` could become `Coord` etc).

A.4.2 Export to VTK

Exporting (apart from animations) requires that `draw_displaced_mesh` has been run first, and the mesh has been returned. For example, this can be done for a displaced 3D-solid:

```
mesh = draw_displaced_mesh(Edof,Coord,Dof,4,a,vM)
```

Now the displaced mesh is saved as a variable. Now `export_vtk` is run to save the data as a `.vtk`-file. First the filename is given, then the mesh, according to:

```
export_vtk('exported_mesh', mesh)
```

If multiple meshes are to be exported and visualized as the same object in ParaView, a list can be given like this:

```
export_vtk('exported_mesh', [mesh1, mesh2, mesh3])
```

Exporting any type of mesh is supported, but it is recommended to export meshes from `draw_displaced_mesh`, as this function is optimized for creating a mesh suitable for export.

A.5 Examples

In order to aid in following the examples, the lines are numbered according to the actual lines in the source code. The examples can be found in the CALFEM for Python source code (see `exv1.py - exv5.py` under examples). All of the examples relies on importing of the following packages at the start:

```
8  import numpy as np
9  import calfem.core as cfc
10 import calfem.vedo as cfv
```

A.5.1 Example 1: Spring

This is the example from Chapter 5.1. It is a simple spring model with three elements in a single row. They are connected to four nodes, one of the nodes at the ends is fixed and the node at the other end has a force applied. The springs have differing stifnesses. The undeformed and deformed meshes are visualized in the same figure, then animated in a different figure.

To begin with, the coordinates, and degrees of freedom are defined, along with elements.

```
coord = np.array([
    [0,0,0],
    [0.5,0,0],
    [1,0,0],
    [1.5,0,0]
])
```

```
dof = np.array([
    [1],
    [2],
    [3],
    [4]
])
```

```
edof = np.array([
    [1, 2],
    [2, 3],
    [3, 4]
])
```

Next, material parameters (stiffness) are defined.

```
k = 1000 #N/m
```

```
ep = [3*k, k, 8*k]
```

Global stiffness matrix is created and assembled.

```
ndof = dof.shape[0]*dof.shape[1] # Number of dofs
nel = edof.shape[0] # Number of elements
K = np.zeros([ndof,ndof])
for i in range(nel):
    Ke = cfc.spring1e(ep[i])
    K = cfc.assem(edof[i],K,Ke)
```

Global force vector is defined, along with boundary conditions. The system of equations is solved.

```
f = np.zeros([ndof,1])
f[3,0] = 500 #Newton
```

```
bcPrescr = np.array([1])
a,r = cfc.solveq(K, f, bcPrescr)
```

Now for the visualization, first the undeformed mesh is added. This is followed by the displaced mesh, offset by 20cm in the y-direction. Some text annotations are added to indicate stiffness and force. The displaced mesh is returned and then exported.

```
cfvv.figure(1,flat=True) # 2D-mode
cfv.draw_mesh(edof,coord,dof,1)
mesh = cfv.draw_displaced_mesh(edof,coord,dof,1,a,offset=[0,0.2,0],rende
    ↪  r_nodes=True)
cfv.add_text_3D('k=3 kN/m',[0.15,-0.1,0],size=0.03)
cfv.add_text_3D('k=1 kN/m',[0.65,-0.1,0],size=0.03)
cfv.add_text_3D('k=8 kN/m',[1.15,-0.1,0],size=0.03)
cfv.add_text_3D('F_x =500 N',[1.55,-0.02,0],size=0.03)

# Export the mesh
cfv.export_vtk('export/exv1/exv1', mesh)
```

The visualization is also animated with 20 steps to achieve a smooth animation. Then the visualization is started, the results can be seen in Figure A.1.

```
cfv.figure(2)
steps = 20
cfv.add_text(f'Looping bewteen undef. & def. state w/ {steps}
    ↪  steps',pos='top-middle')
cfv.animation(edof,coord,dof,1,a,loop=True,steps=20,dt=0,export=True,fil
    ↪  e='export/exv1/anim/exv1')

#Start CALFEM-vedo visualization
cfv.show_and_wait()
```

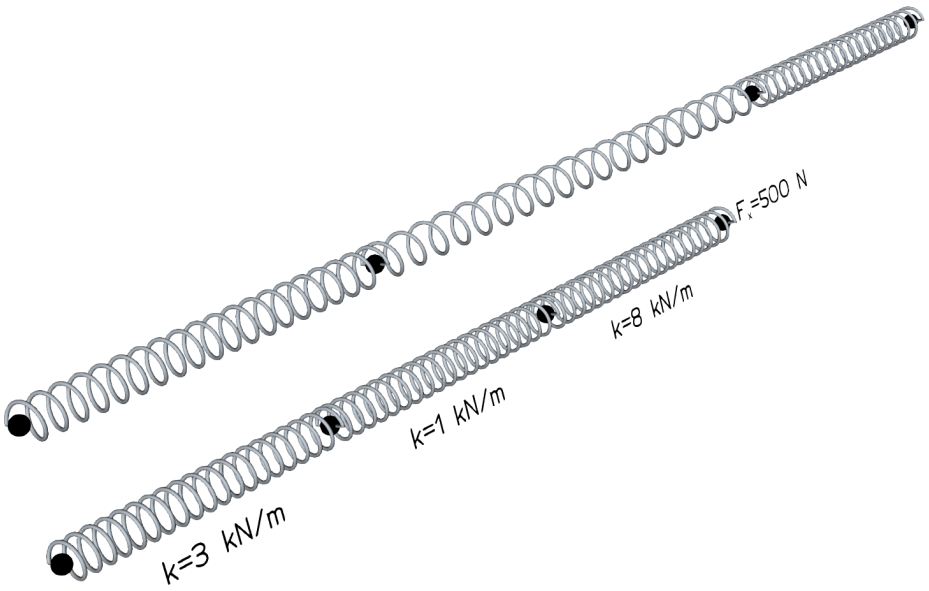


Figure A.1: The resulting spring visualization for example 1

A.5.2 Example 2: Truss

This is the example from Chapter 5.2. It is a truss bridge model consisting of both beam and bar elements. The bridge is 3m wide, 3m tall and 18m in length. It is modeled using symmetry, menaing the length of the model is 9m. Horizontal and vertical elements are mostly beams, bars are placed diagonally to resist deformation in tension, and transverse at the top. The degrees of freedom at the end are fixed. Transverse beams at the bottom are loaded by a uniform load from a concrete deck, and a point load is applied in the transverse direction at the top in the middle of the bridge.

The model is visualized by rendering undeformed and deformed meshes in the same figure, applying a transparency to the undeformed mesh. The deformed mesh are color mapped by the least favorable normal stress (from both normal force and moment when applicable). The moment diagrams for the beams are also plotted along the undeformed beams. Some measurements are added. The meshes are then animated in a separate figure, and exported.

This example contains both bars and beams. They will be handled separately, using different `edof`:s, but sharing `coord` and `dof`. First, the common global matrices are defined:

```
12 coord = np.array([
13     [0, 0, 0],
14     [3, 0, 0],
15     [6, 0, 0],
16     [9, 0, 0],
17     [3, 3, 0],
18     [6, 3, 0],
19     [9, 3, 0],
20     [0, 0, 3],
21     [3, 0, 3],
22     [6, 0, 3],
23     [9, 0, 3],
24     [3, 3, 3],
25     [6, 3, 3],
26     [9, 3, 3]
27 ])
```

```
29 dof = np.array([
30     [1, 2, 3, 4, 5, 6],
31     [7, 8, 9, 10, 11, 12],
32     [13, 14, 15, 16, 17, 18],
33     [19, 20, 21, 22, 23, 24],
34     [25, 26, 27, 28, 29, 30],
35     [31, 32, 33, 34, 35, 36],
36     [37, 38, 39, 40, 41, 42],
37     [43, 44, 45, 46, 47, 48],
38     [49, 50, 51, 52, 53, 54],
39     [55, 56, 57, 58, 59, 60],
40     [61, 62, 63, 64, 65, 66],
41     [67, 68, 69, 70, 71, 72],
42     [73, 74, 75, 76, 77, 78],
43     [79, 80, 81, 82, 83, 84]
44 ])
```

Now the `edof` for beams. Then number of nodes, degrees of freedom, and elements are calculated. Coordinates are then extracted:

```
46 edof_beams = np.array([
47     '''Left side'''
48     [1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12],
49     [7,  8,  9,  10, 11, 12, 13, 14, 15, 16, 17, 18],
50     [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24],
51     [1,  2,  3,  4,  5,  6,  25, 26, 27, 28, 29, 30],
52     [7,  8,  9,  10, 11, 12, 25, 26, 27, 28, 29, 30],
53     [13, 14, 15, 16, 17, 18, 31, 32, 33, 34, 35, 36],
```

```
    [19, 20, 21, 22, 23, 24, 37, 38, 39, 40, 41, 42],
    [25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36],
    [31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42],
    '''Right side'''
    [43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54],
    [49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60],
    [55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66],
    [43, 44, 45, 46, 47, 48, 67, 68, 69, 70, 71, 72],
    [49, 50, 51, 52, 53, 54, 67, 68, 69, 70, 71, 72],
    [55, 56, 57, 58, 59, 60, 73, 74, 75, 76, 77, 78],
    [61, 62, 63, 64, 65, 66, 79, 80, 81, 82, 83, 84],
    [67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78],
    [73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84],
    '''Bottom'''
    [1,  2,  3,  4,  5,  6,  43, 44, 45, 46, 47, 48],
    [7,  8,  9,  10, 11, 12, 49, 50, 51, 52, 53, 54],
    [13, 14, 15, 16, 17, 18, 55, 56, 57, 58, 59, 60],
    [19, 20, 21, 22, 23, 24, 61, 62, 63, 64, 65, 66]
])

nnode = np.size(coord, axis = 0)
ndof = np.size(dof, axis = 0)*np.size(dof, axis = 1)
nel_beams = np.size(edof_beams, axis = 0)

ex_beams,ey_beams,ez_beams = cfc.coordxtr(edof_beams,coord,dof)
```

Next, orientations of beams and material parameters. Then the global stiffness and force vectors are created and forces declared:

```
eo = np.array([
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],

    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1],

    [-1, 0, 0],
    [-1, 0, 0],
    [-1, 0, 0],
    [-1, 0, 0]
    ])
```

```
E = 210000000                # Pa
v = 0.3
G = E/(2*(1+v))              # Pa

#HEA300-beams
A_beams = 11250*0.000001     # m^2
A_web_beams = 2227*0.000001 # m^2
Iy = 63.1*0.000001           # m^4
hy = 0.29*0.5                # m
Iz = 182.6*0.000001          # m^4
hz = 0.3*0.5                 # m
Kv = 0.856*0.000001          # m^4
ep_beams = [E, G, A_beams, Iy, Iz,
 ↪  Kv]

K = np.zeros([ndof,ndof])
f = np.zeros([ndof,1])

# 1 kN point load in z-direction
 ↪  at symmetry (top)
f[38] = 1000 # N
# Transverse beams at y=0
# Dead load of 3x3 m concrete, t =
 ↪  0.2 m, 25 kN/m^2
q = 25*1000*0.2*3 # N/m
eq = [0,-q1,0,0]
```

The global matrices are assembled, first with only beams. If the beam is not loaded with uniform load (<18), no uniform load is included.

```
for i in range(nel_beams):
    if i < 18:
        Ke = cfc.beam3e(ex_beams[i], ey_beams[i], ez_beams[i], eo[i],
        ↪  ep_beams)
        K = cfc.assem(edof_beams[i],K,Ke)
    else:
        Ke, fe = cfc.beam3e(ex_beams[i], ey_beams[i], ez_beams[i],
        ↪  eo[i], ep_beams, eq)
        K, f = cfc.assem(edof_beams[i],K,Ke,f,fe)
```

Bar elements are not going to be added to the model. The element matrix and corresponding coordinates are created manually.

```
edof_bars = np.array([
    [13, 14, 15, 25, 26, 27],
    [19, 20, 21, 31, 32, 33],
    [55, 56, 57, 67, 68, 69],
    [61, 62, 63, 73, 74, 75],
    [25, 26, 27, 67, 68, 69],
    [31, 32, 33, 73, 74, 75],
    [37, 38, 39, 79, 80, 81]
])

ex_bars = np.array([
    [6,3],
    [9,6],
    [6,3],
    [9,6],
    [3,3],
    [6,6],
    [9,9],
])
```

```
ey_bars = np.array([
    [0,3],
    [0,3],
    [0,3],
    [0,3],
    [3,3],
    [3,3],
    [3,3],
])

ez_bars = np.array([
    [0,0],
    [0,0],
    [3,3],
    [3,3],
    [0,3],
    [0,3],
    [0,3],
])
```

Number of elements are calculated, material parameters are defined. Then the global stiffness matrix is assembled using bars as well. Then the system of equations is solved.

```
nel_bars = np.size(edof_bars, axis = 0)

#Solid 100mm circular bars
A_bars = 0.05*np.pi*np.pi    # m^2

ep_bars = [E, A_bars]

for i in range(nel_bars):
    Ke = cfc.bar3e(ex_bars[i], ey_bars[i], ez_bars[i], ep_bars)
    K = cfc.assem(edof_bars[i],K,Ke)
```

```
191
192 bcPrescr = np.array([1, 2, 3, 4, 5, 6, 19, 22, 23, 24, 37, 40, 41, 42,
    ↪  43, 44, 45, 46, 47, 48, 61, 64, 65, 66, 79, 82, 83, 84])
193
194 a,r = cfc.solveq(K, f, bcPrescr)
```

Now the beam section forces are calculated. This is done by dividing the beam into 12 segments, this means the forces are calculated at 13 points along each beam (2 at ends, 11 along its length). The forces are saved as separate variables.

```
196 ed_beams = cfc.extractEldisp(edof_beams,a)
197
198 # Number of points along the beam
199 nseg=13 # 13 points in a 3m long beam = 250mm long segments
200
201 es_beams = np.zeros((nel_beams*nseg,6))
202 edi_beams = np.zeros((nel_beams*nseg,4))
203 eci_beams = np.zeros((nel_beams*nseg,1))
204
205 for i in range(nel_beams):
206     if i < 18:
207         es_beams[nseg*i:nseg*i+nseg,:], edi_beams[nseg*i:nseg*i+nseg,:],
          ↪  eci_beams[nseg*i:nseg*i+nseg,:] =
          ↪  cfc.beam3s(ex_beams[i],ey_beams[i],ez_beams[i],eo[i],ep_beam
          ↪  s,ed_beams[i],[0,0,0,0],nseg)
208     else:
209         es_beams[nseg*i:nseg*i+nseg,:], edi_beams[nseg*i:nseg*i+nseg,:],
          ↪  eci_beams[nseg*i:nseg*i+nseg,:] = cfc.beam3s(ex_beams[i],ey_
          ↪  beams[i],ez_beams[i],eo[i],ep_beams,ed_beams[i],eq,nseg)
210
211 N_beams = es_beams[:,0]
212 Vy = es_beams[:,1]
213 Vz = es_beams[:,2]
214 T = es_beams[:,3]
215 My = es_beams[:,4]
216 Mz = es_beams[:,5]
```

The above step is repeated for the bars. These are not (and cannot be) segmented.

```
218 ed_bars = cfc.extractEldisp(edof_bars,a)
219
220 es_bars = np.zeros((nel_bars,1))
221
222 for i in range(nel_bars):
223     es_bars[i,:] =
      ↪  cfc.bar3s(ex_bars[i],ey_bars[i],ez_bars[i],ep_bars,ed_bars[i])
224
225 N_bars = es_bars
```

Now the normal stresses are calculated. Steiner's formula is used. Since The beams have two moment-components contributing to normal stress, the normal force direction needs to be checked to get the least favorable stress. If the beam is in tension from

normal force, the moment-components are added as tension as well. This stress will only occur at one corner of the beam, where both moments are positive. If the normal force is compressing, the negative moments are used instead. Bars have no moments.

```
227 normal_stresses_beams = np.zeros(nel_beams*nseg)
228
229 # Stress calculation based on element forces
230 for i in range(nel_beams*nseg):
231     if N_beams[i] < 0:
232         normal_stresses_beams[i] = N_beams[i]/A_beams -
            ↪ np.absolute(My[i]/Iy*hz) - np.absolute(Mz[i]/Iz*hy)
233     else:
234         normal_stresses_beams[i] = N_beams[i]/A_beams +
            ↪ np.absolute(My[i]/Iy*hz) + np.absolute(Mz[i]/Iz*hy)
235
236 normal_stresses_bars = np.zeros(nel_bars)
237
238 for i in range(nel_bars):
239     # Calculate least favorable normal stress
240     normal_stresses_bars[i] = N_bars[i]/A_bars
```

In order to color code the beams that are loaded, the `eq_els` matrix is created. It contains element numbers and is used as input to `draw_mesh` along with `eq`. It works in a similar way to `bcPrescr` and `bcVal`, see CALFEM for Python documentation. Both undeformed and deformed meshes are then added to the figure.

```
242 eq_els = np.array([[18],[19],[20],[21]])
243
244 eq = np.zeros([nel_beams,4])
245
246 # For color-coding elements with force applied
247 eq[eq_els[0,:]] = eq
248 eq[eq_els[1,:]] = eq
249 eq[eq_els[2,:]] = eq
250 eq[eq_els[3,:]] = eq
251
252 bcPrescr = np.transpose(bcPrescr)
253
254 cfv.draw_mesh(edof_beams,coord,dof,5,nseg=nseg,alpha=0.2, eq_els=eq_els,
    ↪ eq=eq[eq_els])
255 beams = cfv.draw_displaced_mesh(edof_beams,coord,dof,5,a,normal_stresses
    ↪ _beams/1000000,nseg=nseg,scalar_title='Max normal stress
    ↪ [MPa]')
```

In order to visualize the bars, the element matrix for these need updating to include all 6 degrees of freedom for each node. This is a current limitation to the vedo visualization module. An alternative solution is to use two different `Dof`-matrices instead.

```
260 edof_bars = np.array([
261     [13, 14, 15, 16, 17, 18, 25, 26, 27, 28, 29, 30],
262     [19, 20, 21, 22, 23, 24, 31, 32, 33, 34, 35, 36],
263     [55, 56, 57, 58, 59, 60, 67, 68, 69, 70, 71, 72],
```

```
    [61, 62, 63, 64, 65, 66, 73, 74, 75, 76, 77, 78],
    [25, 26, 27, 28, 29, 30, 67, 68, 69, 70, 71, 72],
    [31, 32, 33, 34, 35, 36, 73, 74, 75, 76, 77, 78],
    [37, 38, 39, 40, 41, 42, 79, 80, 81, 82, 83, 84]
])
```

Now the bars are added to the figure. For the color mapping to work correctly for deformed bars, the minimum/maximum scalar values have to be used as input. This makes sure that the colormap scale is the same for both element types.

```
cfv.draw_mesh(edof_bars,coord,dof,2,alpha=0.2)
vmin, vmax = np.min(normal_stresses_beams), np.max(normal_stresses_beams)
bars = cfv.draw_displaced_mesh(edof_bars,coord,dof,2,a,normal_stresses_b
    ↪ ars/1000000,vmin=vmin,vmax=vmax,scalar_title='Max normal stress
    ↪ [MPa]')
```

Next, the moments are extracted for the 4 beams loaded with uniform load. The diagrams are then added to the figure along with rulers for measurements.

```
Mz_beams = np.zeros((4,nseg))
eci_beams_upd = np.zeros((4,nseg))
for i in range(Mz_beams.shape[0]):
    Mz_beams[i] = Mz[(18+i)*nseg:(18+i)*nseg+nseg]
    eci_beams_upd[i] =
     ↪ np.transpose(eci_beams[(18+i)*nseg:(18+i)*nseg+nseg])

cfv.eldia(ex_beams[18:22],ey_beams[18:22],ez_beams[18:22],Mz_beams/1000,
    ↪ eci_beams_upd,label='M_x
    ↪ [kNm]')
cfv.add_rulers()
```

The animation is (and has to be) done in a separate figure. 20 steps are used, this should be adapted to the displacement. The elements need to be added in two steps. The scalar bar title is used as input to the animation, this is optional but allows ParaView to name the scalars correctly after export. Then the visualization is run and the meshes from the first figure are exported. Note that the meshes are exported in a list, which exports them to the same file.

```
cfv.figure(2)
steps = 20
cfv.add_text(f'Looping from undef. to def. state w/ {steps}
    ↪ steps',pos='top-middle')
cfv.animation(edof_beams,coord,dof,5,a,normal_stresses_beams/1000000,nse
    ↪ g=nseg,dt=125,steps=20,export=True,file='export/exv2/anim/exv2_beams
    ↪ ',scalar_title='Max normal stress
    ↪ [MPa]')
cfv.animation(edof_bars,coord,dof,2,a,normal_stresses_bars/1000000,nseg=
    ↪ nseg,dt=125,steps=20,export=True,file='export/exv2/anim/exv2_bars',s
    ↪ calar_title='Max normal stress
    ↪ [MPa]',vmax=vmax,vmin=vmin)
```

```
288 cfv.add_scalar_bar('Max normal stress [MPa]')
289
290 #Start CALFEM-vedo visualization
291 cfv.show_and_wait()
292
293 #Export the mesh
294 cfv.export_vtk('export/exv2/exv2', [beams,bars])
```

The resulting visualization for the first figure can be seen in Figure A.2

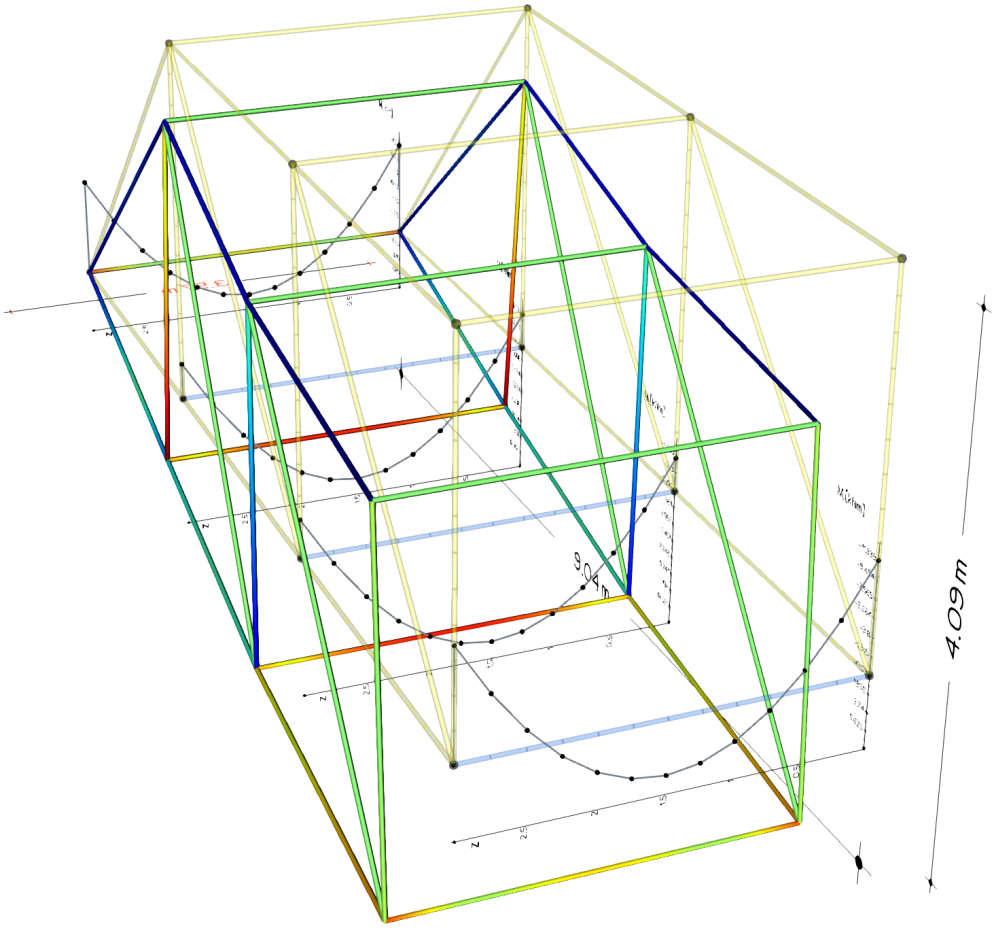


Figure A.2: The resulting truss visualization for example 2

A.5.3 Example 3: Flow

This is the example from Chapter 5.3. This is a heat flow problem in a 400x400mm concrete element, 1m tall. The top and bottom surfaces are 20 degrees, and two arbitrary elements inside the model have heat supplied.

This example plots geometry, undeformed and deformed meshes, then an animation. The undeformed mesh also shows nodes with boundary conditions and elements with force (heat supply). Heat flux using both scalars and vectors are visualized, along with a contour plot of the temperature. Two different ways to add axes to a figure are also shown.

For flow problems, the mesh cannot be 'deformed'. But for the sake of consistency and avoiding confusion as to which functions are used, it is referred to as deformed. Whenever a 'deformed' mesh is visualized, `draw_displaced_mesh` is used.

This example requires a few extra CALFEM modules at the start. After these are defined, the geometry is created using the CALFEM `geometry` module.

```
11 import geometry as cfg
12 import mesh as cfm
13 import utils as cfu
14
15 l = 1
16 h = 0.4
17 w = 0.4
18
19 n_el_x = 4
20 n_el_y = 4
21 n_el_z = 10
22
23 g = cfg.Geometry()
24
25 g.point([0, 0, 0], 0)
26 g.point([w, 0, 0], 1)
27 g.point([w, l, 0], 2)
28 g.point([0, l, 0], 3)
29 g.point([0, 0, h], 4)
30 g.point([w, 0, h], 5)
31 g.point([w, l, h], 6)
32 g.point([0, l, h], 7)
33
34 g.spline([0, 1], 0, el_on_curve = n_el_x)
35 g.spline([1, 2], 1, el_on_curve = n_el_z)
36 g.spline([2, 3], 2, el_on_curve = n_el_x)
37 g.spline([3, 0], 3, el_on_curve = n_el_z)
38 g.spline([0, 4], 4, el_on_curve = n_el_y)
39 g.spline([1, 5], 5, el_on_curve = n_el_y)
40 g.spline([2, 6], 6, el_on_curve = n_el_y)
41 g.spline([3, 7], 7, el_on_curve = n_el_y)
42 g.spline([4, 5], 8, el_on_curve = n_el_x)
43 g.spline([5, 6], 9, el_on_curve = n_el_z)
44 g.spline([6, 7], 10, el_on_curve = n_el_x)
45 g.spline([7, 4], 11, el_on_curve = n_el_z)
46
47 marker_bottom = 40
48 marker_top = 41
49 marker_fixed_left = 42
50 marker_back = 43
51 marker_fixed_right = 44
52 marker_front = 45
53
54 g.structuredSurface([0, 1, 2, 3], 0, marker=marker_bottom)
55 g.structuredSurface([8, 9, 10, 11], 1, marker=marker_top)
56 g.structuredSurface([0, 4, 8, 5], 2, marker=marker_fixed_left)
57 g.structuredSurface([1, 5, 9, 6], 3, marker=marker_back)
58 g.structuredSurface([2, 6, 10, 7], 4, marker=marker_fixed_right)
59 g.structuredSurface([3, 4, 11, 7], 5, marker=marker_front)
60
```

```
61  g.structuredVolume([0,1,2,3,4,5], 0)
```

Now the mesh is created using the CALFEM `mesh` module. Coordinates are extracted.

```
63  el_type = 5
64  dofs_per_node = 1
65  elSizeFactor = 0.01
66
67  coord, edof, dof, bdof, elementmarkers = cfm.mesh(g, el_type,
    ↪  elSizeFactor, dofs_per_node)
68  ex, ey, ez = cfc.coordxtr(edof, coord, dof)
69
70  nnode = np.size(coord, axis = 0)
71  ndof = np.size(dof, axis = 0)*np.size(dof, axis = 1)
72  nel = np.size(edof, axis = 0)
```

Material parameters and forces are defined. The global matrices are then assembled. The elements numbers for the elements with heat flow are found by first visualizing the undeformed mesh.

```
74  Lambda = 1.4 # Thermal conductivity for concrete
75  D = np.identity(3)*Lambda
76
77  K = np.zeros((ndof,ndof))
78
79  f = np.zeros([ndof,1])
80  eq = np.zeros([nel,1])
81
82  eq_els = np.array([[70],[89]])
83
84  eq[eq_els[0]] = 30000
85  eq[eq_els[1]] = -30000
86
87  for eltopo, elx, ely, elz, eqs in zip(edof, ex, ey, ez, eq):
88      Ke,fe = cfc.flw3i8e(elx, ely, elz, [2], D, eqs)
89      cfc.assem(eltopo, K, Ke, f, fe)
```

Boundary conditions are defined and the problem is solved.

```
91  bc = np.array([],'i')
92  bcVal = np.array([],'i')
93
94  bc, bcVal = cfu.apply_bc_3d(bdof, bc, bcVal, marker_fixed_left, 20.0)
95  bc, bcVal = cfu.apply_bc_3d(bdof, bc, bcVal, marker_fixed_right, 20.0)
96
97  T,r = cfc.solveq(K, f, bc, bcVal)
```

Displacements are extracted and heat fluxes calculated.

```
99  ed = cfc.extract_eldisp(edof,T)
100
```

```
es = np.zeros((8,3,nel))
edi = np.zeros((8,3,nel))
eci = np.zeros((8,3,nel))

for i in range(nel):
    es[0:8,:,i], edi[0:8,:,i], eci[0:8,:,i] =
    ↪  cfc.flw3i8s(ex[i],ey[i],ez[i],[2],D,ed[i])
```

Flux vectors are calculated using the average from the gauss points. Total element flux is calculated to use as a scalar colormap.

```
flux = np.zeros((nel,3));
flux_tot = np.zeros((nel,1));
for i in range(nel):
    flux[i,:] = [np.average([es[:,0,i]]), np.average([es[:,1,i]]),
    ↪  np.average([es[:,2,i]])]

    flux_tot[i] = np.sqrt(flux[i,0]**2 + flux[i,1]**2 + flux[i,2]**2)
```

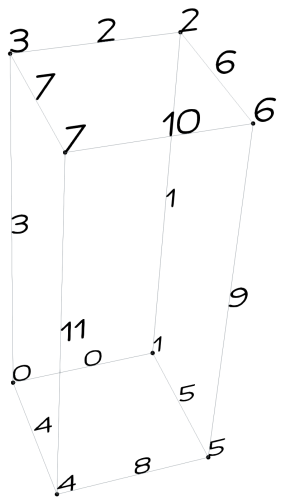
Points, lines, and surfaces are now extracted and the geometry is visualized.

```
# For simple point coordinates of geometry
points = g.points

# For lines with point connectivity
lines = g.curves

# For surfaces with line connectivity
surfaces = g.surfaces

cfv.draw_geometry(points,lines,surfaces)
```

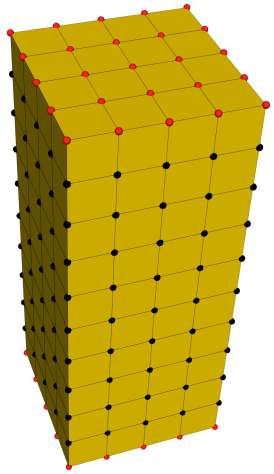
Next, the undeformed mesh is visualized along with forces and boundary conditions. The forces (heat supply) are applied in elements inside the model. These can be revealed by either applying a transparency or by clicking adjacent elements and then pressing the ↓- or x-key (see Section A.6).

```
cfv.figure(2)
cfv.draw_mesh(edof,coord,dof,3,alpha=1,scale=0.005,bcPrescr=bc,
↪  bc=bcVal, eq_els=eq_els, eq=eq[eq_els])
```

Now the first figures are visualized (see Figure A.3). Figure 3 is a deformed mesh using wireframe and color mapped using the total heat flux scalars. In the same figure, the heat flux vectors are added. Figure 4 is a deformed mesh showing the temperature distribution at the nodes. The amount of colors are reduced to 5 (default is 256) in order to produce a contour effect. Both figures also use some form of 3D axes, showing how this can be done in different ways. The resulting visualizations can be seen in Figure A.4

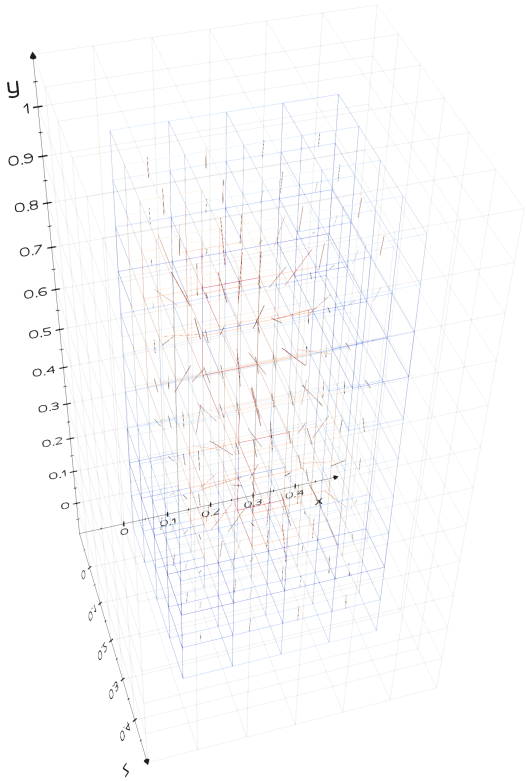


(a) Geometry

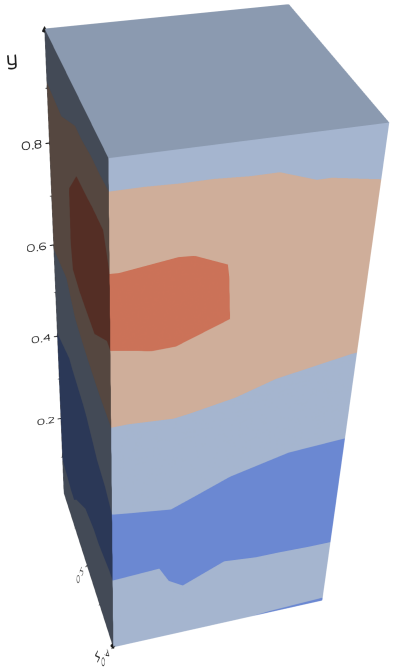


(b) Mesh

Figure A.3: Resulting geometry and mesh visualizations for example 3



(a) Heat flux



(b) Temperature

Figure A.4: Resulting flux and temperature visualizations for example 3

```
131 cfv.figure(3)
132 cfv.draw_displaced_mesh(edof,coord,dof,3,scalars=flux_tot,colormap='cool
    ↪  warm',wireframe=True)
133 cfv.add_axes(xrange=[-0.1,0.5], yrange=[-0.1,1.1], zrange=[-0.1,0.5])
134
135 cfv.add_scalar_bar('Heat flux [W/m^2]')
136 cfv.elflux(ex,ey,ez,flux,colormap='coolwarm')
137
```

```
138 cfv.figure(4)
139 mesh = cfv.draw_displaced_mesh(edof,coord,dof,3,scalars=ed,colormap='coo
    ↪ lwarm',colors=5,scalar_title='Temp.
    ↪ [C]')
140 cfv.add_mesh_axes(mesh)
141 cfv.add_scalar_bar('Temp. [C]')
```

The color distribution is then animated in the last figure. The visualization is started. Finally, the mesh with temperature scalars is exported.

```
146 cfv.figure(5)
147 cfv.animation(edof,coord,dof,3,scalars=ed,dt=250,steps=20,colormap='cool
    ↪ warm',colors=5,export=True,file='export/exv3/anim/exv3',scalar_title
    ↪ ='Temp.
    ↪ [C]')
148 cfv.add_scalar_bar('Temp. [C]')
149
150 #Start CALFEM-vedo visualization
151 cfv.show_and_wait()
152
153 # Export the mesh
154 cfv.export_vtk('export/exv3/exv3', mesh)
```

A.5.4 Example 4: Solid

This is the example from Chapter 5.4. It demonstrates import from CALFEM for MATLAB, for the source code of the MATLAB file, see Appendix B. It's a 5m long beam consisting of 600 elements. The cross-section is 300x400mm. The beam is fixed at both ends.

Two analyses are done for this beam, one static analysis using self-weight and an eccentric load an edge on top on the beam, representing a line load. The line load is approximately half of the beam length and applied in the middle along it. Element von Mises stresses and principal stresses are calculated and visualized in a deformed state.

The second analysis is a modal analysis using both stiffness and mass of the beam to find the lowest eigenfrequency, along with the corresponding eigenmode. The eigenmode is visualized as deformed mesh.

Before visualizing results from both analyses, the undeformed mesh is visualized along with forces and boundary conditions for the static analysis. Results from both analyses are also animated.

This example uses some additional utilities imported at the start. Then MATLAB results are imported and coordinates extracted.

```
11 import vedo_utils as cfvu
12
```

```
13 edof,coord,dof,a,ed,bc,f_dofs,Stress_tensors,vM,lamb,eig =
   ↪ cfvv.import_mat('exv4',['edof','coord','dof','a','ed','bc','force_do
   ↪ fs','Stress_tensors','vM','lambda','eig'])
14
15 ex,ey,ez = cfc.coordxtr(edof,coord,dof)
```

The first eigenmode is chosen, then total deformations for it are calculated in order to map it as scalars later.

```
17 eigenmode = 0 # Choose what eigenmode to display in figure 5/6
18
19 ndof = np.size(dof, axis = 0)*np.size(dof, axis = 1)
20 ncoord = np.size(coord, axis = 0)
21 nel = np.size(edof, axis = 0)
22
23 mode_a = np.zeros((nel, 1))
24 tot_deform = np.zeros(8)
25 for i in range(nel):
26         coords = cfvu.get_coord_from_edof(edof[i,:],dof,4)
27         for j in range(8):
28                 deform =
                   ↪ cfvu.get_a_from_coord(coords[j],3,eig[:,eigenmode])
29                 tot_deform[j] = np.sqrt(deform[0]**2 + deform[1]**2 +
                   ↪ deform[2]**2)
30
31         mode_a[i,:] = np.average(tot_deform)
32
33 Freq=np.sqrt(lamb[eigenmode]/(2*np.pi))
```

Principal stresses are calculated based on the stress tensors imported from MATLAB.

```
37 ps_val = np.zeros((nel,3))
38 ps_vec = np.zeros((nel,3,3))
39 for i in range(nel):
40     ps_val[i,:], ps_vec[i,:,:] = np.linalg.eig(Stress_tensors[:,:,i])
```

Total displacements from the static analysis are calculated in order to visualize them as scalars later.

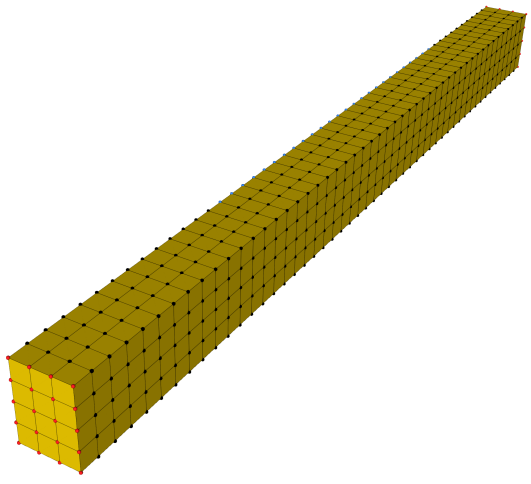
```
44 upd_ed = np.zeros((nel,8))
45 for i in range(nel):
46     upd_ed[i,0] = np.sqrt( ed[i,0]**2 + ed[i,1]**2 + ed[i,2]**2 )
47     upd_ed[i,1] = np.sqrt( ed[i,3]**2 + ed[i,4]**2 + ed[i,5]**2 )
48
49     upd_ed[i,2] = np.sqrt( ed[i,6]**2 + ed[i,7]**2 + ed[i,8]**2 )
50     upd_ed[i,3] = np.sqrt( ed[i,9]**2 + ed[i,10]**2 + ed[i,11]**2 )
51
52     upd_ed[i,4] = np.sqrt( ed[i,12]**2 + ed[i,13]**2 + ed[i,14]**2 )
53     upd_ed[i,5] = np.sqrt( ed[i,15]**2 + ed[i,16]**2 + ed[i,17]**2 )
54
55     upd_ed[i,6] = np.sqrt( ed[i,18]**2 + ed[i,19]**2 + ed[i,20]**2 )
56     upd_ed[i,7] = np.sqrt( ed[i,21]**2 + ed[i,22]**2 + ed[i,23]**2 )
```

Boundary conditions are converted, a force vector is created in order to color code nodes where the eccentric line load is applied.

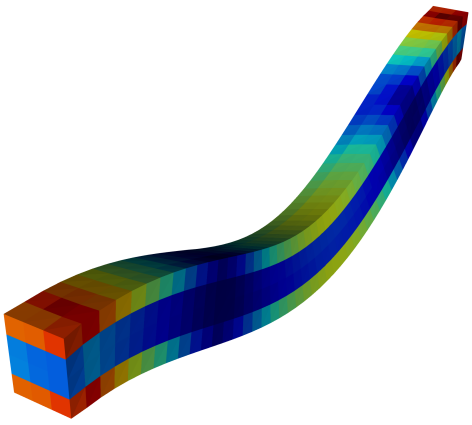
```
58 bcPrescr = bc
59 bc = np.zeros((np.size(bc[:,0]),1))
60 f = -5000*np.ones((np.size(f_dofs[:,0]),1))
```

Now for the first two visualizations. In this example, all figures are visualized one-by-one. This means the user has to close them before the next one is rendered. The first figure is the undeformed mesh with forces and boundary conditions for the static analysis. The second plot is the deformed mesh for the static analysis, with von Mises (vM) stresses. The visualizations can be seen in Figure A.5.

```
62 # First plot, undeformed mesh
63 cfv.figure(1)
64
65 cfv.draw_mesh(edof,coord,dof,4,scale=0.005,bcPrescr=bcPrescr[:,0],bc=bc[
   ↪  :,0],fPrescr=f_dofs[:,0],f=f[:,0])
66 cfv.add_text('Undeformed mesh + Forces & BCs for static analysis')
67 cfv.show_and_wait()
68
69 # Second plot, deformed mesh with element stresses
70 cfv.figure(2)
71
72 scalefact = 3 #deformation scale factor
73 static = cfv.draw_displaced_mesh(edof,coord,dof,4,a,vM/1000000,def_scale
   ↪  =scalefact,scalar_title='von Mises
   ↪  [MPa]')
74
75 cfv.add_text('Static analysis: self-weight & ecc. vertical load',
   ↪  pos='top-left')
76 cfv.add_text(f'Deformation scalefactor: {scalefact}',pos='top-right')
77 cfv.add_scalar_bar('von Mises [MPa]')
78 cfv.show_and_wait()
```



(a) Undeformed mesh with forces and boundary conditions for static analysis

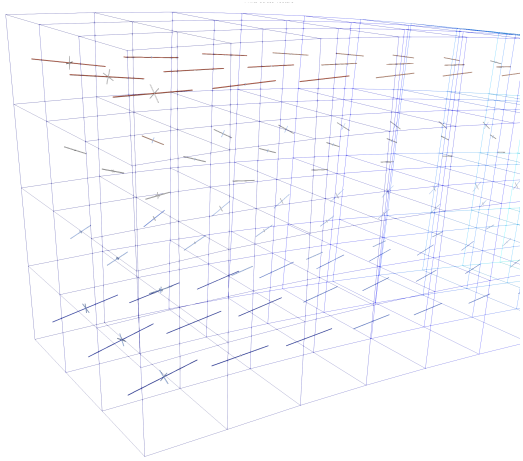


(b) Deformed mesh for static analysis with von Mises stresses

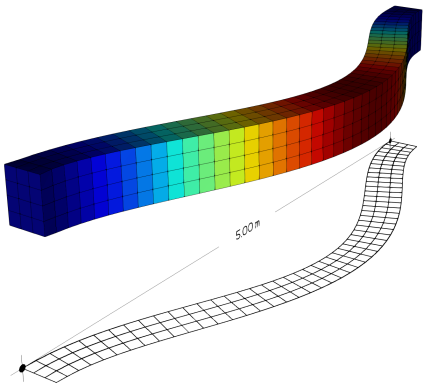
Figure A.5: Resulting mesh visualizations for example 4

In the third plot, the static analysis results are animated. In the fourth, the principal stresses are plotted inside the elements (Figure A.6a). This is done with a deformed wireframe mesh in the same figure, in order to see the outlines of the elements. This wireframe is also color mapped with the total deformation from this analysis.

```
80  # Third plot, animation of figure 2
81  cfv.figure(3)
82  scalefact = 3 #deformation scale factor
83
84  cfv.animation(edof,coord,dof,4,a,vM/1000000,def_scale=scalefact,export=T
    ↪  rue,file='export/exv4/anim/exv4_static',scalar_title='von Mises
    ↪  [MPa]')
85
86  cfv.add_text('Static analysis: self-weight & ecc. vertical load',
    ↪  pos='top-left')
87  cfv.add_text(f'Deformation scalefactor: {scalefact}',pos='top-right')
88
89  cfv.add_scalar_bar('von Mises [MPa]')
90
91  #Start Calfem-vedo visualization
92  cfv.show_and_wait()
93
94  # Fourth plot, principal stresses for static analysis
95  cfv.figure(4)
96
97  # Return the mesh for export
98  cfv.draw_displaced_mesh(edof,coord,dof,4,a,upd_ed*1000,wireframe=True)
99  cfv.elprinc(ex,ey,ez,ps_val/1000000,ps_vec,ed,colormap='coolwarm',unit='
    ↪  MPa')
100 cfv.add_scalar_bar('Deformation [mm]')
101 cfv.add_text('Static analysis',pos='top-left')
102 cfv.add_text('Deformation scalefactor: 1',pos='top-right')
103 cfv.add_text('Princ. stress vectors',pos='top-middle')
104
105 #Start Calfem-vedo visualization
106 cfv.show_and_wait()
```



(a) Principal stresses for static analysis



(b) Deformed mesh for modal analysis

Figure A.6: Resulting stress and mesh visualizations for example 4

The last two plots are for the modal analysis. The first shows the deformed mesh with the total element displacement color mapped, with a large deformation scale factor applied (Figure A.6b). The last figure is an animation of this. The deformed meshes from both analyses are then exported as separate files.

```
# Fifth plot, first mode from eigenvalue analysis
cfv.figure(5)

scalefact = 100 #deformation scale factor
modal = cfv.draw_displaced_mesh(edof,coord,dof,4,eig[:,eigenmode],mode_a
    *1000,def_scale=scalefact,lines=True,scalar_title='Tot. el.
    displacement [mm]')
cfv.add_text(f'Modal analysis: {eigenmode+1}st mode',pos='top-left')
cfv.add_text(f'Frequency: {round(Freq[0],2)} Hz')
cfv.add_text(f'Deformation scalefactor: {scalefact}',pos='top-right')
cfv.add_scalar_bar('Tot. el. displacement [mm]')
cfv.add_projection(plane='xz',rulers=True)
cfv.show_and_wait()

# Sixth plot, animation of figure 5
cfv.figure(6)

cfv.add_text(f'Modal analysis: {eigenmode+1}st mode',pos='top-left')
cfv.add_text(f'Frequency: {round(Freq[0],2)} Hz')
cfv.add_text(f'Deformation scalefactor: {scalefact}',pos='top-right')

cfv.animation(edof,coord,dof,4,eig[:,eigenmode],mode_a*1000,def_scale=sc
    alefact,negative=True,scalar_title='Tot. el. displacement
    [mm]',export=True,file='export/exv4/anim/exv4_modal')

cfv.add_scalar_bar('Tot. el. displacement [mm]')

#Start CALFEM-vedo visualization
cfv.show_and_wait()

# Export the two meshes
cfv.export_vtk('export/exv4/exv4_static', static)
cfv.export_vtk('export/exv4/exv4_modal', modal)
```

A.5.5 Example 5: Plate

This is the example from Chapter 5.5. It's a 3x3m plate consisting of 900 elements. It is loaded with self-weight. Two edges opposite each other are fixed. Along the other edges the degree of freedom in the z-direction is prescribed in the middle, representing columns. Element von Mises stresses are calculated and mapped as element scalars on the deformed mesh. The undeformed mesh is also visualized, along with animating the deformed mesh.

First, since the `platrs`-function is currently not supported, it is converted to python for later use.

```
# Adaptation of platrs from version 3.4 for MATLAB
# This function is not in the Python version currently
```

```
# Only es is included as it's the only needed output
def platrs(ex,ey,ep,D,ed):
    Lx=ex[2]-ex[0]
    Ly=ey[2]-ey[0]
    t=ep[0]

    D=((t**3)/12)*D

    A1=D[1,1]/2/Ly
    A2=D[0,0]/2/Lx
    A3=D[0,1]/2/Ly
    A4=D[0,1]/2/Lx
    A5=D[2,2]/2/Ly
    A6=D[2,2]/2/Lx
    A7=4*D[2,2]/Lx/Ly

    B1=6*D[1,1]/Ly/Ly/Ly
    B2=6*D[0,0]/Lx/Lx/Lx
    B3=-3*D[1,1]/Ly/Ly
    B4=3*D[0,0]/Lx/Lx
    B5=D[0,1]/Lx/Ly

    mx=A3*(-ed[1]-ed[4]+ed[7]+ed[10])+A2*(ed[2]-ed[5]-ed[8]+ed[11])
    my=A1*(-ed[1]-ed[4]+ed[7]+ed[10])+A4*(ed[2]-ed[5]-ed[8]+ed[11])
    mxy=A6*(ed[1]-ed[4]-ed[7]+ed[10])+A5*(-ed[2]-ed[5]+ed[8]+ed[11])+A7*
     ↪  (ed[0]-ed[3]+ed[6]-ed[9])

    m1=0.5*(mx+my)+np.sqrt(0.25*(mx-my)**2+mxy**2)
    m2=0.5*(mx+my)-np.sqrt(0.25*(mx-my)**2+mxy**2)
    alfa=0.5*180/np.pi*np.arctan2(mxy,(mx-my)/2)

    vx=B5*(-ed[1]+ed[4]-ed[7]+ed[10])+B4*(ed[2]+ed[5]+ed[8]+ed[11])+B2*(
     ↪  -ed[0]+ed[3]+ed[6]-ed[9])
    vy=B3*(ed[1]+ed[4]+ed[7]+ed[10])+B5*(ed[2]-ed[5]+ed[8]-ed[11])+B1*(-
     ↪  ed[0]-ed[3]+ed[6]+ed[9])

    es=np.transpose(np.array([mx, my, mxy, vx, vy]))

    return es
```

Next, the mesh is created, starting by defining the width and thickness of each plate element. Then the `coord` and `dof` matrices are created using loops.

```
d=0.1
t=0.05

ncoord_x = 30+1
ncoord_y = 30+1

ncoord_init = ncoord_x*ncoord_y

coord = np.zeros([ncoord_init,2])
row = 0

for y in range(ncoord_y):
    for x in range(ncoord_x):
```

```
64          coord[row,:] = [x*d,y*d]
65          row = row+1
66  ncoord = np.size(coord,0)
67
68  dof = np.zeros([ncoord,3])
69
70  it = 1
71  dofs = [0,1,2]
72  for row in range(ncoord):
73      for col in dofs:
74          dof[row,col] = it
75          it = it + 1
76
77  ndof = np.size(dof,0)*np.size(dof,1)
78
79  nel_x = (ncoord_x-1)
80  nel_y = (ncoord_y-1)
81
82  edof = np.zeros((nel_x*nel_y,4*3))
83  bc = np.zeros((ncoord_y*2*3*2-12,1))
```

Next, the `edof` matrix is assembled. Boundary conditions along edges are created in the same loop.

```
85  x_step = 1
86  y_step = ncoord_x
87
88  it = 0
89  bc_it = 0
90  node = 0
91
92  for row in range(nel_y):
93      for el in range(nel_x):
94
95          edof[it,0:3] = dof[node,:]
96
97          if el == 0:
98              bc[bc_it,0] = int(dof[node,0])
99              bc_it = bc_it + 1
100             bc[bc_it,0] = int(dof[node,1])
101             bc_it = bc_it + 1
102             bc[bc_it,0] = int(dof[node,2])
103             bc_it = bc_it + 1
104
105         node = node+x_step
106         edof[it,3:6] = dof[node,:]
107         if el == nel_x-1:
108             bc[bc_it,0] = int(dof[node,0])
109             bc_it = bc_it + 1
110             bc[bc_it,0] = int(dof[node,1])
111             bc_it = bc_it + 1
112             bc[bc_it,0] = int(dof[node,2])
113             bc_it = bc_it + 1
114
115         node = node+y_step;
116         edof[it,6:9] = dof[node,:]
```

```
117
118         if el == nel_x-1:
119             bc[bc_it,0] = int(dof[node,0])
120             bc_it = bc_it + 1
121             bc[bc_it,0] = int(dof[node,1])
122             bc_it = bc_it + 1
123             bc[bc_it,0] = int(dof[node,2])
124             bc_it = bc_it + 1
125
126         node = node-x_step
127         edof[it,9:12] = dof[node,:]
128
129         if el == 0:
130             bc[bc_it,0] = int(dof[node,0])
131             bc_it = bc_it + 1
132             bc[bc_it,0] = int(dof[node,1])
133             bc_it = bc_it + 1
134             bc[bc_it,0] = int(dof[node,2])
135             bc_it = bc_it + 1
136
137         if el == nel_x-1:
138             node = node-y_step+2
139         else:
140             node = node+x_step-y_step
141
142         it = it+1
```

Now material parameters and loads are defined.

```
144 edof = np.int_(edof)
145
146 nnode = np.size(coord, axis = 0)
147 ndof = np.size(dof, axis = 0)*np.size(dof, axis = 1)
148 nel = np.size(edof, axis = 0)
149
150 ep=[t]
151
152 E=25*1000000000
153 v=0.2
154 eq=-250*1000
155
156 D = cfc.hooke(1,E,v);
```

Coordinates are extracted and global matrices are assembled.

```
159 ex, ey = cfc.coordxtr(edof,coord,dof)
160
161 K = np.zeros((ndof,ndof))
162 f = np.zeros((ndof,1))
163
164 for eltopo, elx, ely in zip(edof, ex, ey):
165     Ke,fe = cfc.platre(elx, ely, ep, D, eq)
166     # Transposing fe due to error in platre
167     fe = np.transpose(fe)
168     cfc.assem(eltopo, K, Ke, f, fe)
```

Extra boundary conditions are applied, representing columns in the middle of remaining edges. Global matrices are assembled.

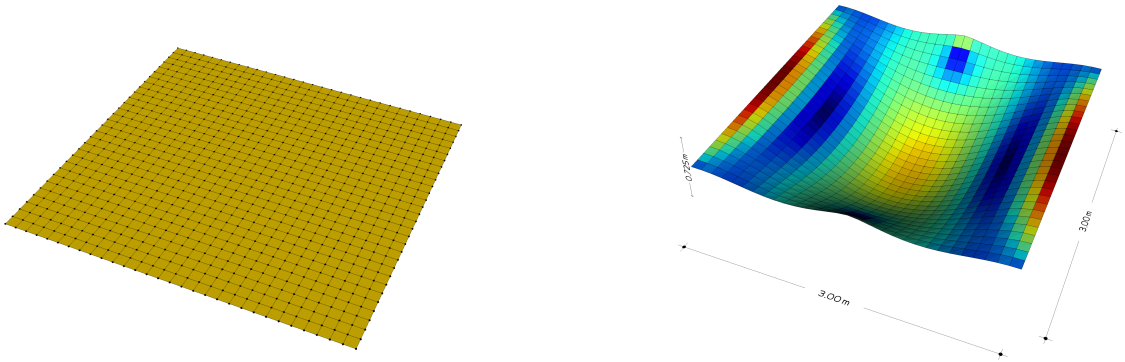
```
170 bc = bc[:,0]
171 bc = bc.astype(np.int64)
172 extra_bcs = np.array([15*3+1,945*3+1])
173 bc = np.concatenate((bc, extra_bcs))
174
175 a,r = cfc.solveq(K, f, bc)
```

Displacements are extracted, element stresses are calculated using the converted `platrs` function. After this, von Mises (`vM`) stresses are calculated for each element.

```
177 ed = cfc.extract_eldisp(edof,a)
178 es = np.zeros((nel,5))
179
180 for i in range(nel):
181     es[i,:] = platrs(ex[i],ey[i],ep,D,ed[i])
182
183 vM = np.zeros((nel,1))
184
185 for i in range(nel):
186     sigma_xx = es[i,0]/t
187     sigma_yy = es[i,1]/t
188     sigma_xy = es[i,2]/t
189
190     sigma_1 = (sigma_xx+sigma_yy)/2 + np.sqrt(
    ↪  ((sigma_xx+sigma_yy)**2)/4 + sigma_xy**2)
191     sigma_2 = (sigma_xx+sigma_yy)/2 - np.sqrt(
    ↪  ((sigma_xx+sigma_yy)**2)/4 + sigma_xy**2)
192
193     vM[i] = np.sqrt(sigma_1**2 -sigma_1*sigma_2 + sigma_2**2)
```

Now for the visualization. The undeformed mesh is rendered in its own figure. Then the deformed mesh, along with rulers and some text, is rendered. The resulting visualizations can be seen in Figure A.7.

```
194 cfv.draw_mesh(edof,coord,dof,6,scale=0.005)
195
196 cfv.figure(2)
197 def_scale = 5
198 mesh = cfvv.draw_displaced_mesh(edof,coord,dof,6,a,scale=0.002,t=t,def_s⌋
    ↪  cale=def_scale,values=vM/1000000)
199 cfv.add_rulers()
200 cfv.add_scalar_bar('Max. von Mises [MPa]')
201 cfv.add_text(f'Defomation scale: {def_scale}')
202
203 #Start CALFEM-vedo visualization
204 cfv.show_and_wait()
```



(a) Mesh

(b) Deformed mesh

Figure A.7: Resulting mesh visualizations for example 5

The deformed mesh is now animated with 20 steps, 250ms between. The deformed mesh is then exported.

```
206 cfv.figure(3)
207
208 cfv.animation(edof,coord,dof,6,a,vM/1000000,def_scale=def_scale,dt=250,s
    ↪  teps=20,export=True,file='export/exv5/anim/exv5',scalar_title='von
    ↪  Mises [MPa]')
209 cfv.add_scalar_bar('von Mises [MPa]')
210 cfv.add_text(f'Defomation scale: {def_scale}')
211
212 #Start CALFEM-vedo visualization
213 cfv.show_and_wait()
214
215 # Export the mesh
216 cfv.export_vtk('export/exv5/exv5', mesh)
```

A.6 Interaction

Vedo includes some keyboard shortcuts for interacting with the model by default. Some potentially useful shortcuts are listed in the table below. The global shortcuts affect the whole rendering. The actor shortcuts can be used on all or a single actor (if one is highlighted). These can function inconsistently as all actors don't support all shortcuts. For instance, a cutter tool doesn't support surface meshes, which means it will not work for plate elements.

Key(s)	**Description**	**Use**
	Global	
`5/6`	Change background color	Cycle through background colors
`+/-`	Change axis style	Change axis representation (bottom left)
`A`	Toggle anti-aliasing	Smooth edges (can slow rendering)
`D`	Toggle depth-peeling	Hides actors behind transparent actors
`a`	Toggle actor mode	Disables camera (only actor interaction)
`r`	Reset camera	Resets camera zoom & rotation
`C`	Print camera settings	For debugging
`S`	Save screenshot	Save as .png in working directory
`q`	Exit interaction	Disables camera
`Esc`	Abort execution	Stop animation loops
	Actor	
←/→	Reduce/Increase opacity	Change transparency
`w/s`	Wireframe/solid rendering	Wireframe shows only lines
`l`	Toggle edges	Show/hide lines
`x`	Toggle visibility	Hide actor
`X`	Cutter tool	Cut an actor (partially hide)
`1/2`	Change color	Cycle through actor colors
`k`	Change lighting style	Remove shadows etc.
`K`	Change shading style	Change actor appearance
`n`	Show surface normals	Add vectors representing surface normals

A.7 Function reference

A.7.1 Main functions

Drawing geometry

```
draw_geometry(points=None, lines=None, surfaces=None,
 ↪  scale=0.05)
```

Displays geometry as defined by the `geometry` module. Input variables `points`, `lines` and `surfaces` are as defined in the module. If `surfaces` are given, `lines` and `points` are required. `Points` are always required.

Input		Return
`points`:	Geometry points	-
`lines`:	Geometry lines	
`surfaces`:	Geometry surfaces	
`scale`:	Radius for lines. Point radius is 1.5x this scale	

Drawing mesh

```
draw_mesh(edof, coord, dof, element_type, scale=0.02,
 ↪  alpha=1, render_nodes=True, color='yellow',
 ↪  offset=[0,0,0], bcPrescr=None, bc=None, bc_color='red',
 ↪  fPrescr=None, f=None, f_color='blue6', eq_els=None,
 ↪  eq=None, spring=True, nseg=2)
```

Draws undeformed mesh element-wise. Supports visualization of elements/nodes where forces/boundary conditions are applied, by coloring. **NOTE:** If the model has a large number of elements, it is recommended to use `draw_displaced_mesh` (if possible) instead.

Element types: 1 → Spring, 2 → Bar, 3 → Flow, 4 → Solid, 5 → Beam, 6 → Plate

Input	
`edof`:	Element-wise degrees of freedom
`coord`:	Node coordinates
`dof`:	Degrees of freedom by node
`element_type`:	Element type
`scale`:	Radius for beam/bar/spring. Node radius is 1.5x this scale
`alpha`:	Transparency. 0 is invisible, 1 is fully visible
`render_nodes`:	If set to `True`, nodes will be rendered
`color`:	Color of element, supports text strings and RGB, see here
`offset`:	Offset elements in any direction

bcPrescr:	Prescribed boundary conditions
bc:	Boundary conditions
bc_color:	Color for denoting nodes with BCs applied
fPrescr:	Prescribed forces
f:	Forces
f_color:	Color for denoting nodes/elements with forces applied
eq_els:	Elements with forces
eq:	Element forces
spring:	If true, springs are coil springs
nseg:	Number of segments for internal element values for a beam

Return

mesh:	List containing meshes by element

Drawing displaced mesh

```
draw_displaced_mesh(edof, coord, dof, element_type, a=None,
↪  scalars=None, scale=0.02, alpha=1, def_scale=1,
↪  render_nodes=False, color='white', offset=[0,0,0],
↪  only_ret=False, lines=False, wireframe=False,
↪  colormap='jet', colors=256, vmax=None, vmin=None,
↪  scalar_title='scalar', spring=True, nseg=2)
```

Draws deformed mesh using an unstructured grid. Can be color mapped with results at elements and nodes. For flow, solid and plate elements, a single actor is created. For other elements, one actor per element is created, as in draw_mesh.

Element types: 1 → Spring, 2 → Bar, 3 → Flow, 4 → Solid, 5 → Beam, 6 → Plate

For colormapping, different color maps are available. See Vedo documentation at:

https://vedo.embl.es/autodocs/content/vedo/

Input

edof:	Element-wise degrees of freedom
coord:	Node coordinates
dof:	Degrees of freedom by node
element_type:	Element type
a:	Global displacement vector
scalars:	Global displacement vector
scale:	Radius for beam/bar/spring. Node radius is 1.5x this scale
alpha:	Transparency. 0 is invisible, 1 is fully visible
def_scale:	Deformation scalefactor
render_nodes:	If set to True, nodes will be rendered
color:	Color of element, supports text strings and RGB, see here
offset:	Offset elements in any direction

only_ret:	Only return mesh for export
lines:	Draw lines
wireframe:	Wireframe mode, only lines
colormap:	Name of colormap, see here
colors:	No. colors
vmin:	Manual minimum for colormap
vmax:	Manual maximum for colormap
scalar_title:	Scalar title for export
spring:	If true, springs are coil springs
nseg:	Number of segments for internal element values for a beam

Return

mesh:	List of meshes (1/2/5) **or** merged mesh (3/4/6)

Animation

```
animation(edof, coord, dof, element_type, a=None,
 ↪  scalars=None, steps=10, loop=False, negative=False,
 ↪  dt=50, animate_colormap=True, scale=0.02, alpha=1,
 ↪  def_scale=1, only_export=False, export=False,
 ↪  file='anim/CALFEM_anim', colormap='jet', colors=256,
 ↪  vmax=None, vmin=None, scalar_title='scalar',
 ↪  spring=True, nseg=2)
```

Creates a timeline between undeformed and deformed states and animates it. Number of steps and rate is given by the user. This function relies on `draw_displaced_mesh`, and runs this function to create each step. Exporting of animations is built-in. The following modes are available:

$$\texttt{Default}: \quad 0 \rightarrow 1$$

$$\texttt{Loop}: \quad 0 \rightarrow 1 \rightarrow 0$$

$$\texttt{Negative}: \quad 0 \rightarrow 1 \rightarrow 0 \rightarrow (-1) \rightarrow 0$$

Element types: 1 → Spring, 2 → Bar, 3 → Flow, 4 → Solid, 5 → Beam, 6 → Plate

NOTE: Animating color maps for spring, bar, and beam elements currently not supported.

Input

edof:	Element-wise degrees of freedom
coord:	Node coordinates
dof:	Degrees of freedom by node
element_type:	Element type
a:	Global displacement vector
scalars:	Scalars

steps:	Total number of steps in animation
loop:	Loop mode
negative:	Negative mode
dt:	Time difference between steps
animate_colormap:	Include colormap in animation
scale:	Element/node scale
alpha:	Transparency. 0 is invisible, 1 is fully visible
def_scale:	Deformation scalefactor
only_export:	Export to ParaView, no CALFEM animation
export:	If True, will export animation as keyframes for ParaView
file:	Path of folder/file to export to
colormap:	Name of colormap, see here
colors:	No. colors in colormap
vmax:	Scalar maximum
vmin:	Scalar minimum
scalar_title:	Scalar title for export
spring:	If true, springs are coil springs
nseg:	No. segments for beams

Return
-

Selecting figure

```
figure(fig, bg='white', flat=False, hover=False)
```

Selects which figure to add elements to. Everything in this Section that is run after this adds objects to the specified figure, including things such as text. If a black background is used, colors for some objects are inverted. If 2D-mode is used, interaction by hovering is used automatically.

Input		Return
fig:	Figure number	-
bg:	Background color	
flat:	2D-mode	
hover:	Get node/element info by hovering	

Running visualization

```
show_and_wait()
```

Show all figures created, starts VTK rendering event loop.

Input	Return
-	-

A.7.2 Import/Export

Importing from Matlab

```
import_mat(file, list=None)
```

Imports data from '.mat' files. If any form of `Edof` is detected, its formatting is automatically converted to CALFEM for Pythons version.

Input		Return
`file`:	Path to file	-
`list`:	List of variables to import, and order of them	

Exporting to VTK/ParaView

```
export_vtk(file, meshes)
```

Exports a mesh or many meshes to a single file. If many meshes are provided, they will be automatically merged.

Input		Return
`file`:	Path to file	-
`meshes`:	Mesh/list of meshes to export	

A.7.3 Miscellaneous functions

Adding a scalar bar

```
add_scalar_bar(label, pos=[0.8,0.05], font_size=24,
↪  color='black')
```

Adds a scalar bar for a displaced mesh with a colormap. **NOTE:** If a displaced mesh is added to the same figure as a geometry/mesh, the displaced mesh has to be added first for scalar bars to work.

Input		Return
`label`:	Text label	-
`pos`:	Text label pos, see `add_text`	
`font_size`:	Text size	
`color`:	Text color	

Adding text

```
add_text(text, color='black', pos='top-middle', size=1)
```

Add 2D text in the rendering window. Position can be a descriptive string like 'top-left' or a 2D vector like (0.2,0.8).

Input		Return
`text`:	Text	-
`color`:	Text color	
`pos`:	Position in rendering window	
`size`:	Text size	

Adding 3D text

```
add_text_3D(text, pos=(0,0,0), color='black', size=1,
↪  alpha=1)
```

Add 3D text in the rendered scene.

Input		Return
`text`:	Text	-
`pos`:	Position in 3D	
`color`:	Text color	
`size`:	Text size	
`alpha`:	Transparency	

Adding a projection

```
add_projection(color='black', plane='xy', offset=-1,
↪  rulers=False)
```

Adds a projection of model to a plane. Recommended to run after main functions.

Input		Return
`color`:	Color of projection	-
`plane`:	Plane to project to (xy/xz/yz)	
`offset`:	Offset plane	
`rulers`:	Automatically add rulers if True	

Adding rulers

```
add_rulers()
```

Automatically adds rulers to figure. Recommended to run after main functions.

Input	Return
-	-

Adding axes

```
add_axes(xrange=[0,1], yrange=[0,1], zrange=[0,1],
 ↪  xtitle='x', ytitle='y', ztitle='z', htitle='',
 ↪  xyGrid=True, yzGrid=True, zxGrid=True, xyGrid2=False,
 ↪  yzGrid2=False, zxGrid2=False, xyGridTransparent=True,
 ↪  yzGridTransparent=True, zxGridTransparent=True,
 ↪  xyGrid2Transparent=True, yzGrid2Transparent=True,
 ↪  zxGrid2Transparent=True, numberOfDivisions=10)
```

Adds custom axes to a figure.

Input

`xrange`:	Min/max for x-axis
`yrange`:	Min/max for y-axis
`zrange`:	Min/max for z-axis
`xtitle`:	Title for x-axis
`ytitle`:	Title for y-axis
`ztitle`:	Title for z-axis
`htitle`:	Main title
`xyGrid`:	Grid pattern on xy-plane
`yzGrid`:	Grid pattern on yz-plane
`zxGrid`:	Grid pattern on xz-plane
`xyGrid2`:	Secondary grid pattern on xy-plane
`yzGrid2`:	Secondary grid pattern on yz-plane
`zxGrid2`:	Secondary grid pattern on xz-plane
`xyGridTransparent`:	Transparency for `xyGrid`
`yzGridTransparent`:	Transparency for `yzGrid`
`zxGridTransparent`:	Transparency for `zyGrid`
`xyGrid2Transparent`:	Transparency for `xyGrid2`
`yzGrid2Transparent`:	Transparency for `yzGrid2`
`zxGrid2Transparent`:	Transparency for `zxGrid2`
`numberOfDivisions`:	No. markers per axis

Return

-

Adding axes for a mesh

```
add_mesh_axes(mesh)
```

Automatically adds axes to figure based on a mesh. Recommended to run after main functions.

Input		Return
`mesh`:	Mesh for adding axes to	-

Section force diagram for beams

```
eldia(ex, ey, ez, es, eci, dir='y', scale=1, thickness=5,
 ↪  alpha=1, label='y', invert=True)
```

Draws a section force diagram along a beam in 3D.

Input		Return
ex:	x-coordinates	-
ey:	y-coordinates	
ez:	z-coordinates	
es:	Section force	
eci:	Points along beam	
dir:	Direction if diagram y-axis (x/y/z)	
scale:	Diagram scale	
thickness:	Line/point scale	
alpha:	Transparency	
label:	Label diagram	
invert:	Invert diagram	

Vectors

```
elflux(ex, ey, ez, vec, ed=None, scale=.1, colormap='jet',
 ↪  unit='W/m^2')
```

Draws vectors mid-element in 3D. Support deformed meshes if `ed` is used.

Input		Return
ex:	x-coordinates	-
ey:	y-coordinates	
ez:	z-coordinates	
vec:	Vectors	
ed:	Element displacements	
scale:	Vector scale	
colormap:	Colormap for vectors	
unit:	Vector unit (output when clicking/hovering)	

Principal stresses

```
elprinc(ex, ey, ez, val, vec, ed=None, scale=.1,
 ↪  colormap='jet', unit='Pa')
```

Draws principle stresses mid-element in 3D. Support deformed meshes if `ed` is used.

Input		Return
ex:	x-coordinates	-
ey:	y-coordinates	
ez:	z-coordinates	
val:	Eigenvalues	
vec:	Eigenvectors	
ed:	Element displacements	
scale:	Vector scale	
colormap:	Colormap for vectors	
unit:	Vector unit (output when clicking/hovering)	

A.7.4 Utilities

These functions require `import vedo_utils`.

Get coordinates for element

```
get_coord_from_edof(edof_row, dof, element_type)
```

Get element coordinates for single element based on type.

Input		Return	
edof_row:	Element number	coords:	Coordinates
dof:	Degrees of freedom by node		
element_type:	Element type		

Get displacements for node

```
get_a_from_coord(coord_row_num, num_of_deformations, a,
↪  scale=1)
```

Get displacements (from global vector) for a single node.

Input	
coord_row_num:	Node number
num_of_deformations:	Degrees of freedom per node
a:	Global displacements
scale:	Deformation scale

Return	
dx:	Displacement x-axis
dy:	Displacement y-axis
dz:	Displacement z-axis

Appendix B: Solid example

MATLAB code

This Appendix contains the MATLAB source code for the example in Chapter 5.4.

```
% CALFEM Vedo Visualization example (exv4)
% Author: Andreas Åmand



% --- Creating mesh ---

d=0.1; % Elements are 100 mm x 100 mm x 100 mm

% No. elements per direction
nel_x = 50;
nel_y = 4;
nel_z = 3;
nel = nel_x*nel_y*nel_z;

% No. nodes per direction
nnode_x = nel_x+1;
nnode_y = nel_y+1;
nnode_z = nel_z+1;
nnode = nnode_x*nnode_y*nnode_z;

% --- Creates Coord matrix ---

coord = zeros(nnode,3);
row = 1;
for z = 0:nnode_z-1
    for y = 0:nnode_y-1
        for x = 0:nnode_x-1
            coord(row,:) = [x*d,y*d,z*d];
            row = row+1;
        end
    end
end

% --- Creates Dof matrix ---

dof = zeros(nnode,3);
it = 1;
for row = 1:nnode
    for col = 1:3
        dof(row,col) = it;
        it = it + 1;
    end
end
```

```
ndof = size(dof,1)*size(dof,2);

% --- Creates Edof and Boundary Condition matrices ---
% Boundary conditions: DoFs at x = 0m & x = 50m have a displacement of 0

x_step = 1; % Next node in x-direction
y_step = nnode_x; % Next node in y-direction
z_step = (y_step)*nnode_y; % Next node in z-direction

it = 1; % Element number for loops (used as index in edof)
bc_it = 1; % Iteration for bc (used as index in bc)
force_dof_it = 1; % Iteration for point load dofs (used as index in
↪  force_dofs)
node = 1; % For keeping track of node

edof = zeros(nel_x*nel_y*nel_z,8*3+1);
bc = zeros(nnode_y*nnode_z*2*3,2);
force_dofs = zeros(25+1,1); % for saving dofs to apply point loads to
for col = 0:nel_z-1 % Loops through z-axis
    node = 1+z_step*col;
    for row = 0:nel_y-1 % Loops through y-axis
        for el = 0:nel_x-1 % Loops through x-axis
            edof(it,1) = it; % Element number, first row in Edof

            % --- First node ---
            edof(it,2:4) = dof(node,:); % Dofs for first element node
            if el == 0  % If element is at x = 0, save bc
                bc(bc_it,1) = dof(node,1);
                bc_it = bc_it + 1;
                bc(bc_it,1) = dof(node,2);
                bc_it = bc_it + 1;
                bc(bc_it,1) = dof(node,3);
                bc_it = bc_it + 1;
            end

            % --- Second node ---
            node = node+x_step; % Gets node number
            edof(it,5:7) = dof(node,:); % Gets dofs for node
            if el == nel_x-1 % If element is at x = 5, save bc
                bc(bc_it,1) = dof(node,1);
                bc_it = bc_it + 1;
                bc(bc_it,1) = dof(node,2);
                bc_it = bc_it + 1;
                bc(bc_it,1) = dof(node,3);
                bc_it = bc_it + 1;
            end

            % --- Third node ---
            node = node+y_step;
            edof(it,8:10) = dof(node,:);
            if el == nel_x-1
                bc(bc_it,1) = dof(node,1);
                bc_it = bc_it + 1;
                bc(bc_it,1) = dof(node,2);
                bc_it = bc_it + 1;
                bc(bc_it,1) = dof(node,3);
                bc_it = bc_it + 1;
            end
```

```
102
103             % If elements at x = 0 and top row, save y-dofs for later
104             if (col == 0 && row == 3 && ismember(el,(13:38)) == 1)
105                 force_dofs(force_dof_it) = dof(node,2);
106                 force_dof_it = force_dof_it + 1;
107             end
108
109             % --- Fourth node ---
110             node = node-x_step;
111             edof(it,11:13) = dof(node,:);
112             if el == 0
113                 bc(bc_it,1) = dof(node,1);
114                 bc_it = bc_it + 1;
115                 bc(bc_it,1) = dof(node,2);
116                 bc_it = bc_it + 1;
117                 bc(bc_it,1) = dof(node,3);
118                 bc_it = bc_it + 1;
119             end
120
121             % --- Fifth node ---
122             node = node+z_step-y_step;
123             edof(it,14:16) = dof(node,:);
124             if el == 0
125                 bc(bc_it,1) = dof(node,1);
126                 bc_it = bc_it + 1;
127                 bc(bc_it,1) = dof(node,2);
128                 bc_it = bc_it + 1;
129                 bc(bc_it,1) = dof(node,3);
130                 bc_it = bc_it + 1;
131             end
132
133             % --- Sixth node ---
134             node = node+x_step;
135             edof(it,17:19) = dof(node,:);
136             if el == nel_x-1
137                 bc(bc_it,1) = dof(node,1);
138                 bc_it = bc_it + 1;
139                 bc(bc_it,1) = dof(node,2);
140                 bc_it = bc_it + 1;
141                 bc(bc_it,1) = dof(node,3);
142                 bc_it = bc_it + 1;
143             end
144
145             % --- Seventh node ---
146             node = node+y_step;
147             edof(it,20:22) = dof(node,:);
148             if el == nel_x-1
149                 bc(bc_it,1) = dof(node,1);
150                 bc_it = bc_it + 1;
151                 bc(bc_it,1) = dof(node,2);
152                 bc_it = bc_it + 1;
153                 bc(bc_it,1) = dof(node,3);
154                 bc_it = bc_it + 1;
155             end
156
157             % --- Eighth node ---
158             node = node-x_step;
159             edof(it,23:25) = dof(node,:);
```

```
160             if el == 0
161                 bc(bc_it,1) = dof(node,1);
162                 bc_it = bc_it + 1;
163                 bc(bc_it,1) = dof(node,2);
164                 bc_it = bc_it + 1;
165                 bc(bc_it,1) = dof(node,3);
166                 bc_it = bc_it + 1;
167             end
168 
169             % Reset node
170             if el == nel_x-1 % If last element
171                 node = node-z_step-y_step+2;
172             else % Otherwise, first node for next el. = second node for
                ↪  current
173                 node = node+x_step-y_step-z_step;
174             end
175 
176             it = it+1;
177 
178         end
179     end
180 end
181 
182 % --- Creating Global Stiffness & Force matrices ---
183 
184 [ex,ey,ez] = coordxtr(edof,coord,dof,8);
185 
186 ep = [2]; % No. integration points
187 
188 % Material parameters for steel
189 E = 210000000; % Modulus of elasticity [Pa]
190 v = 0.3; % Poisson's ratio
191 D = hooke(4,E,v); % Material matrix
192 
193 % Loads
194 g=9.82; % Gravitational constant
195 rho = 7850; % Density [kg/m^3]
196 eq = [0; -g*rho; 0]; % Distibuted load vector [N/m^3]
197 
198 f = zeros(ndof,1);
199 K = zeros(ndof);
200 for i=(1:nel) % Assembling
201     [Ke,fe] = soli8e(ex(i,:), ey(i,:), ez(i,:), ep, D, eq);
202     [K,f] = assem(edof(i,:), K, Ke, f, fe);
203 end
204 
205 % --- Applying point forces  ---
206 
207 point_force = -5000; % N
208 f(force_dofs) = f(force_dofs) + point_force;
209 
210 
211 
212 % First a linear analysis is done (self-weight + eccentric line load)
213 
214 % --- Solving system of equations & Extracting global displacements  ---
215 
216 a = solveq(K, f, bc);
```

```
ed = extract(edof,a);

% --- Extracting global displacements & Calculating element stresses ---

es = zeros(ep*ep*ep,6,nel);
et = zeros(ep*ep*ep,6,nel);
eci = zeros(ep*ep*ep,3,nel);
for i=(1:nel)
    [es(:,:,i),et(:,:,i),eci(:,:,i)] =
    ↪  soli8s(ex(i,:),ey(i,:),ez(i,:),ep,D,ed(i,:));
end

% --- Calculating Stress Tensors & von Mises ---

Stress_tensors = zeros(3,3,nel);
vM = zeros(nel,1);
for i=(1:nel)
    s_xx = mean(es(:,1,i));
        s_yy = mean(es(:,2,i));
        s_zz = mean(es(:,3,i));
        s_xy = mean(es(:,4,i));
        s_xz = mean(es(:,5,i));
        s_yz = mean(es(:,6,i));

    Stress_tensors(:,:,i) = [s_xx s_xy s_xz; s_xy s_yy s_yz; s_xz s_xy
    ↪  s_zz];
    vM(i) = sqrt( 0.5*((s_xx-s_yy)^2 + (s_yy-s_zz)^2 + (s_xx-s_yy)^2) +
    ↪  3*(s_xy^2 + s_xz^2 + s_yz^2) );
end



% Now an eigenvalue analysis of the model is done

% --- Gauss points from soli8e/soli8s ---

g1=0.577350269189626;
gp(:,1)=[-1; 1; 1;-1;-1; 1; 1;-1]*g1;
gp(:,2)=[-1;-1; 1; 1;-1;-1; 1; 1]*g1;
gp(:,3)=[-1;-1;-1;-1; 1; 1; 1; 1]*g1;

xsi=gp(:,1);  eta=gp(:,2); zet=gp(:,3);

% --- Masses for element ---

m = zeros(8);
for i = (1:8)
    for j = (1:8)
        m(i,j) = (rho*d*d*d/8)*(1+(1/3)*xsi(i)*xsi(j))*(1+(1/3)*eta(i)*e
        ↪  ta(j))*(1+(1/3)*zet(i)*zet(j));
    end
end

% --- Element mass matrix ---

iter_i = 1;
iter_j = 1;

```

```
271 Me = zeros(3*8);
272 for i = (1:3:3*8)
273     iter_j = 1;
274     for j = (1:3:3*8)
275         Me(i,j) = m(iter_i,iter_j);
276         Me(i+1,j+1) = m(iter_i,iter_j);
277         Me(i+2,j+2) = m(iter_i,iter_j);
278         iter_j = iter_j+1;
279     end
280     iter_i = iter_i+1;
281 end
282
283 % --- Global mass matrix ---
284
285 M = zeros(ndof);
286 for i=(1:nel)
287     M = assem(edof(i,:), M, Me);
288 end
289
290 % --- Eigenvalue analysis ---
291
292 [lambda,eig] = eigen(K,M,bc(:,1));
293
294
295
296 % --- Results from both analyses are saved in a .mat-file ---
297
298 save('exv4.mat','coord','dof','edof','bc','force_dofs','a','ed','Stress_
    ↪  tensors','vM','lambda','eig')
```

Appendix C: Survey

Survey of tools for visualization

Survey

Development of functions for visualization in CALFEM for Python

Andreas Åmand
vov15aam@student.lu.se

Note: If you only use/plan to use CALFEM for Matlab, please fill out the form with regards to the Matlab version. The developed tools will hopefully be able to import results from Matlab.

Note: Some functionality is already implemented. However, the survey is also of use do determine where focus on improving functions should be put. Documentation can be found here.

Note: If filled in digitally, it works best in Adobe Acrobat Reader. Printing, filling out & scanning is fine.

1. I work at the division of:					
2. Are you involved in a course that uses CALFEM for teaching?				Yes	No
3. If no above: would you consider using CALFEM for teaching if tools for visualization were improved/added?				Yes	No
4. If you use/were to use CALFEM, which dimensions of elements would be of use for visualizing?			1D	2D	3D
5. 1D element types that would be of use for visualizing	Spring	Bar	Flow	Other	
6. Types of visualization of use for 1D elements	Underformed shape/mesh Displacements Stresses (σ_{11} etc.)		Derformed shape/mesh Scalar values Other		Section forces
7. 2D element types that would be of use for visualizing	Bar	Flow	Beam	Solid	Plate
	Other				
8. Types of visualization of use for 2D elements	Underformed shape/mesh Displacements Principal stresses von Mises stresses		Derformed shape/mesh Scalar values Contour plots Stresses (σ_{11} etc.)		Section forces Isolines Other
9. 3D element types that would be of use for visualizing	Bar	Flow	Beam	Solid	Other
10. Types of visualization of use for 3D elements	Underformed shape/mesh Displacements Principal stresses von Mises stresses		Derformed shape/mesh Scalar values Contour plots Stresses (σ_{11} etc.)		Section forces Isolines Other
11. Would animations of deformations or similar be useful?				Yes	No

Feel free to leave any other comments or suggestions on the next page →

Comments/suggestions: