NTNU
Norwegian University of
Science and Technology

Exploring Finite Element Analysis in a Parametric Environment

Thomas Lunde Villanger
Kristian Nikolai Åland

Civil and Environmental Engineering
Submission date: June 2018
Supervisor: Anders Rönnquist, KT
Co-supervisor: Marcin Luczkowski, IBM

Norwegian University of Science and Technology
Department of Structural Engineering

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

ACCESSIBILITY
Open

MASTER THESIS 2018

SUBJECT AREA:	DATE:	NO. OF PAGES:
Structural Engineering	11.06.2018	xii + 154 + 179

TITLE:

Exploring Finite Element Analysis in a Parametric Environment

Utforsking av elementmetoden i et parametrisk miljø

BY:

Thomas Lunde Villanger and Kristian Nikolai Åland


ABSTRACT:

This thesis is *Exploring Finite Element Analysis in a Parametric Environment*, with the intent of building a functioning Finite Element Analysis (FEA) program within the Grasshopper parametric environment. A motivation for this is to provide tools for designers and architects to roughly and swiftly assess structures within the Grasshopper environment.

In order to attain a deeper understanding of how the Finite Element Method can be implemented in a parametric design environment, some Finite Element Analysis software packages are created to gain some experience with the inner processes of the Finite Element algorithms and to help locate eventual implementation issues.

The results are four functioning programs for calculation of displacements, strains and stresses within truss, beam and shell structures. In addition, analysis is performed on each of the programs to assess their performance in terms of running time and accuracy. To measure accuracy, the software packages has been compared to analytical solutions and a well-established Finite Element Analysis program.

All the created software packages display sensible deformation patterns and are in accordance with the established Finite Element Analysis comparison tool. In terms of running time, the simpler software bundles are executed within satisfactory time limits, but the heavier software bundles struggle with larger structures. In general, the processing parts could benefit from utilization of sparse storage formats and better optimized solving algorithms. The software packages are very close to analytical solutions, with the exception of complicated shell structures. The Shell software would benefit from implementation of more advanced elements, especially for the membrane part of the element.

RESPONSIBLE TEACHER:
Professor Anders Rønnquist

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

”Essentially, all models are wrong, but some are useful.”

— George E. P. Box (1987)

Summary

The purpose of this thesis is to explore Finite Element Analysis (FEA) in the Grasshopper parametric environment, with the aim to provide tools to roughly and quickly assess structural performance. Hopefully, such tools would lead to more readily optimized designs and fewer design corrections needing to be sent between the architect and engineer.

In order to achieve this goal, a theoretical chapter has been dedicated to outlining the basics of the Finite Element Method (FEM). The chapter explains concepts fundamental to FEM and mechanics in general. This includes degrees of freedom, relations between force and displacement, transformation matrix, stiffness matrices, and constitutive relations for beams and shells. In addition, the chapter lightly enters the subjects of higher order shape functions and direct solving by Cholesky Banachiewicz Decomposition.

Based on this theoretical background, four separate programs have been made. The first and most basic program, 2D Truss, was made as an introduction to FEM and the Grasshopper workflow. This program lays the foundations for the more complicated programs, as the general method and process remains the same for all of them. Next, 3D Truss expands the program to three dimensions and undergoes a large refactoring in order to make use of the open-source toolkit for C#, Math.NET. With this, the processing part of the program is markedly faster, although some optimization missteps were made in the pre-processing section. Moving on from trusses, the 3D Beam software saw significant changes because of the leap to moments and rotations. Initially, this program followed much the same process as the other two, but especially calculation of strains and displacements within elements were later altered to make use of displacement fields. The beam software is based on Euler-Bernoulli beam theory. Lastly, the stiffness matrices of the shell program were yet another leap from the previous programs. The shell element is created by combining a Constant Strain Triangle for membrane action and a Morley Triangle for bending. Shell structures requires considerably more degrees of freedom for achieving adequate results, and consequently requires larger systems of equations to be solved. This quickly leads to unacceptably long running times.

The software packages for 2D and 3D Truss structures presents satisfactory results regarding runtimes and accuracy when compared to the analytical solution and the estab-

lished FEA software used as a benchmark. As for 3D Beam and Shell, which are more comprehensive and complex, the results deviate slightly from the benchmark program. However, results converge towards the ”correct” solution in all examples where the results were not already identical or constant. For the 3D Beam software, results are very close to the benchmark software, except for the case of uniformly distributed loads. The Shell software deviates more from the correct solutions as the elements chosen are likely very basic in comparison to the ones used by the benchmark software.

The final software packages mostly work as intended, since deformation patterns and stress distributions are displayed correctly, even though accuracy may at times be lacking. Consequently, the software packages can be used to roughly assess structural deformation behavior and stress localization. For Shells, large jumps in stress concentrations can be a problem because of the rudimentary elements chosen. The problem is alleviated somewhat by increasing the number of elements.

The software packages would greatly benefit from further work on the solver for the system of linear equations, as this was found to be the bottleneck for the runtime. They could also benefit from improvements in terms of ease-of-use, improved color-mapping of stress, uniform load distributions and more advanced boundary conditions.

Sammendrag

Formålet med denne avhandlingen er å utforske elementmetoden i det parametriske miljøet til Grasshopper, med sikte på å lage verktøy for å gjøre kjappe og grove vurderinger av strukturell ytelse. Forhåpentligvis vil dette føre til lettere optimaliserte design og færre designkorrigeringer som må sendes mellom arkitekt og ingeniør.

For å oppnå dette målet har et teoretisk kapittel blitt dedikert til å gi en grunnleggende beskrivelse av elementmetoden. Kapittelet forklarer begreper som er fundamentale for elementmetoden og mekanikk. Det omfatter grader av frihet, forhold mellom kraft og forskyvning, transformasjonsmatriser, stivhetsmatriser og konstitutive relasjoner for bjelker og skall. I tillegg går kapitlet lett inn på temaene for høyere ordens formsfunksjoner og direkte løsning ved Cholesky Banachiewicz faktorisering.

Basert på denne teoretiske bakgrunnen er det laget fire separate programmer. Det første og mest grunnleggende programmet, 2D Truss, ble laget som en introduksjon til FEM og arbeidsflyten i Grasshopper. Dette programmet legger grunnlaget for de mer kompliserte programmene ettersom den generelle metoden og prosessen forblir den samme for alle. Neste program, 3D Truss, utvider fagverksberegningene til tre dimensjoner og gjennomgår en stor refaktorering for å kunne benytte et åpen kilde-verktøy til C#, Math.NET. Med dette er prosesseringsdelen av programmet markant raskere, selv om det ble gjort noen feil i forbehandlingsdelen. Med 3D Beam-programvaren ble det overgang fra staver til bjelker, og det var betydelige endringer på grunn av spranget til moment og rotasjon. I utgangspunktet fulgte dette programmet mye samme prosess som de to foregående, men beregning av tøyning og spenning inne i elementer ble senere endret for å utnytte forskyvningsfelt. Bjelkeprogrammet er basert på Euler-Bernoulli bjelketeori. Til slutt var stivhetsmatrisene til skallprogrammet enda et sprang fra de tidligere programmene. Skallelementet opprettes ved å kombinere en Konstant tøyningstriangel (eng: Constant Strain Triangle) for membrankrefter og en Morley-trekant for bøyningskrefter. Skallstrukturer krever betydelig flere grader av frihet for å oppnå tilstrekkelig nøyaktige resultater, og krever følgelig at det må løses større ligningssett. Dette fører raskt til uakseptabelt lange kjøretider.

Programvarepakker for 2D og 3D Truss-strukturer gir gode resultater når det gjelder kjøretid og nøyaktighet, sammenlignet med den analytiske løsningen og den etablerte FEA-programvaren som brukes som referanse. Når det gjelder 3D Beam og Shell, som er mer omfattende og komplekse, avviker resultatene litt fra referanseprogrammet. Resultatene konvergerer imidlertid til den ”riktige” løsningen i alle eksempler hvor resultatene

ikke allerede var like eller konstante. For 3D Beam-programvaren er resultatene svært nær benchmark-programvaren, bortsett fra tilfelle av jevnt fordelte belastninger. Shell-programvaren avviker mer fra de riktige løsningene da de valgte elementene er ganske grunnleggende.

De endelige programvarepakkene for det meste som ønsket, ettersom deformasjonsmønstre og spenningsfordelinger vises korrekt, selv om nøyaktighet til tider er manglende. Følgelig kan programvarepakkene brukes til å gjøre grove vurderinger av strukturell deformasjonsadferd og spenningslokalisering. For skall kan store hopp i spenningskonsentrasjoner være et problem som følge av at elementene er forholdsvis enkle. Problemet lindres noe ved å øke antallet elementer.

Programvarepakkene vil ha stor nytte av videre arbeid på løsningen av ligningssett, da dette ble funnet å være flaskehalsen for kjøretiden. De kan også dra nytte av forbedringer knyttet til brukervennlighet, fargekartlegging av spenninger, jevnt fordelte laster og mer avanserte randverdibetingelser.

Acknowledgements

We would like to extend a very special thanks to our co-supervisor Marcin Luczkowski who has been very helpful by providing suggestions on a topic for this thesis, how to proceed and where to find valuable reading material. Together with Steinar Hillersøy Dyvik and John Haddal Mork he also held a very timely introductory course for Rhino/Grasshopper and C#.

We would also like to thank our supervisor Nils E. A. Rønnquist who helped us decide on a master's thesis that would be interesting to study, as well as answering any questions we had throughout the semester.

Table of Contents

Definitions

FEA	=	Finite Element Analysis
FEM	=	Finite Element Method
CAD	=	Computer Aided Design
NURBS	=	Non-Uniform Rational Basis-Spline
RHS	=	Right Hand Side (of an equation)
gdof	=	Global Degrees of Freedom
ldof	=	Local Degrees of Freedom
rdof/rgdof	=	Reduced Global Degrees of Freedom
component	=	Grasshopper algorithm
software package/bundle	=	Bundle of components belonging to either 2D Truss, 3D Truss, 3D Beam or 3D Shell
BDC/boundary conditions	=	Support conditions (free or clamped)
completion runtime	=	Time spent to complete an algorithm's designated task (running time is also used)

Chapter 1

Introduction

As parametric design becomes more popular among architects and designers, the advantages of a parametric work environment has become evident. One of the major advantages to parametric design is that changes are rapidly visualized for the user. By relating structures to sets of variables, geometry can easily be altered and tweaked. Given the ease with which designs can change early in the design process, it makes sense to give the designer a basic understanding of how the structure will behave structurally. By considering the implications of structural analysis early in the design process, designs may become more easily optimized in terms of structural performance.

The most popular structural analysis software packages are, by far, the various implementations of Finite Element Analysis (FEA). These programs utilize the Finite Element Method (FEM), which divides the main problem into several minor parts called Finite Elements.

The Finite Element Method (FEM) is a method arising from mainly five groups of papers, (K. and L., 1996). FEM was first coined by R.W. Clough in the 1950s after conducting a vibration analysis of a wing structure (Clough, 2001), but papers contributing to the method were made as early as in 1943, by Richard Courant (Williamson, Jr., 1980). Many details regarding his calculations are lacking, however, so it would be problematic to attribute Courant with the origin of FEM (K. and L., 1996). John Argyris published a series of papers in 1954 which related stresses and strains to loads and displacements and establishes a rectangular panel stiffness matrix. Next is M.J Turner, who claimed that a triangular element holds several advantages of rectangular stiffness matrices. He also derived the stiffness matrix for trusses in global coordinates. Turner was the supervisor of Clough when he was working at Boeing Airplane Company. Clough later made significant contributions to FEM by expanding on Turner's work. Zienkiewicz and Cheung are recognized for applying FEM to problems outside solid mechanics, and published the first textbook on FEM in 1957.

The intent of this thesis is to explore the Finite Element Method for use in a parametric environment, with the aim of providing designers and architects with a tool to quickly and roughly assess their work in a structural manner. To achieve this, an attempt to create some parametric Finite Element Analysis software packages will be made, with the intention of achieving a deeper understanding of the potential problems and opportunities that occurs by combining parametric design with Finite Element Analysis. The software's results do not have to be completely accurate, but should provide an insight into the behavior of the structure before being analyzed in depth by structural engineers. This way the structure would (hopefully) be more feasible from the onset, and require fewer design iterations between the engineer and architect. In turn, this means fewer resources are required in the design phase.

This thesis will approach the finite element method with a numerical mindset. The intention is to adequately explain and implement the FEM, oriented towards a programming perspective rather than the mathematical point of view. As the mathematical approach often can seem over-complicated and hard to apply to real-life applications, this thesis aims to "translate" the mathematical formulations to a numerical and implementable language. The relevant mathematical theory will be presented with an attempt to interpret it numerically, and the intention to put it directly to use. With some background experience in C# the reader might be able to create their own Finite FEA software. Hopefully this thesis could act as a guide to create FEA software for parametric environment.

1.1 Clarifications

Components will in this paper refer to the separate objects or ”boxes” in the Grasshopper environment. They can be compared to individual functions or classes in a computer science analogy. These components perform minor or major tasks, and may have multiple data inputs and give multiple data outputs. One component can easily contain many methods, but a method cannot contain a component.

Methods in this paper refer to a procedure or a function in a larger program written i C#. The methods usually perform a certain task, and can be called as many times as is necessary. Methods may be seen as the code equivalent of a component but usually perform minor tasks.

Where there are given coded examples in the C# language, the curly brackets have been removed for readability reasons, and indentations will in these code snippets indicate which operations belongs where.

Global axes are denoted by capital letters (X,Y,Z) while local axes are lower-cased (x,y,z).

Chapter 2

Theory

2.1 Assumptions

Most of this thesis is concerned with simplified models of reality, by this we assume that materials are elastic and homogeneous. In this ”simple world” we assume that linear theory is sufficient to represent deformations. Linear theory is based on two basic assumptions (Bell, 2013):

1. Small displacements, which means that equilibrium and kinematic compatibility can be based on the undeformed geometry.
2. Linear elasticity, which means that the stress-strain relationship is linear and reversible.

2.2 Degree of Freedom

Degrees of freedom (d.o.fs, dofs or singular dof) are the number of independent nodal displacements that are free to change (Saouma, 1999). The term ”displacements” encompasses both translational and rotational freedom, meaning that a complete node for a beam element would have 6 dofs, as illustrated in Figure 2.1. A three-dimensional truss would need 3 dofs (one for each translation), while a two-dimensional truss only needs 2 dofs. A shell element can be defined in many different ways, but in this paper the Constant Strain Triangle and

Morley Triangle has been combined to form a shell element of 9 dofs, see Ch. 2.6, two dofs in each corner and 1 along each edge.

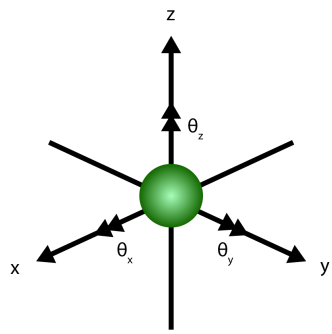
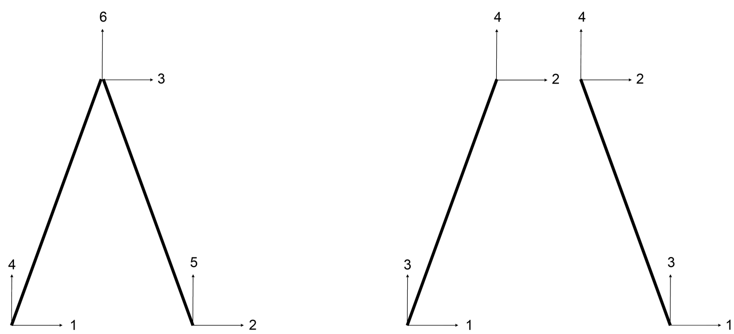


Figure 2.1: Six degrees of freedom

Dofs can be further elaborated by providing boundary conditions. A condition that disallows displacements is called a "clamped" condition. A fully clamped node is called a fixed boundary condition.

The terms global and local degrees of freedom (gdof and ldof), respectively relates dofs to the system as a whole or of each element. The differences are visualized for a 2D system on Fig. 2.2.



(a) Global degrees of freedom in system (b) Local degrees of freedom in element

Figure 2.2: Degrees of freedom in system and element

Reduced degrees of freedom (rdofs) are the number of gdofs remaining after removing any dofs connected to clamped boundary conditions.

2.3 Force-Displacement Relations

The stiffness method used in Finite Element Analysis works by:

1. Constraining all dofs
2. Applying unit displacements at each dof (others remain restrained at zero)
3. Determining the reactions associated with all dofs

In structural problems the reaction forces $\mathbf{R}$ and the nodal displacements $\mathbf{u}$ are related through what is called a *system stiffness matrix* or *global stiffness matrix* as

$$\mathbf{R} = \mathbf{K}\mathbf{u} \tag{Eq. 2.3.1}$$

One of the main challenges here is to establish the $\mathbf{K}$ matrix. This is achieved through determining the *element stiffness matrix* for each element in the structure, and then assemble all of them in the *global stiffness matrix*.

To find the reaction forces $\mathbf{R}$ the displacement vector $\mathbf{u}$ needs to be determined. This can be done by reducing the global stiffness matrix so that all the rows and column corresponding to restrained dofs are removed. This new reduced global stiffness matrix will here be denoted $\mathbf{K}$*. Likewise removing the corresponding entries from the load vector $\mathbf{P}$ gives the reduced load vector $\mathbf{P}^*$. The reduced displacement vector is similarly $\mathbf{u}^*$. By removing these restrained dofs the following is obtained

$$\mathbf{P}^* = \mathbf{K}^*\mathbf{u}^* \tag{Eq. 2.3.2}$$

The structure now is statically determinate, which means it can be solved. However, the $\mathbf{K}$*-matrix is ”ill-conditioned or nearly singular if its determinant is close to zero” (Gavin, 2012). In these cases, $\mathbf{K}$* cannot be easily inverted. This complicates the solving a bit, but it can still be solved as a system of equations, of which there exists many methods for solving.

Solving these systems of equations for the displacement vector $\mathbf{u}^*$ may take some time compared to the other steps of solving the structural problem. For this reason, one of the preferable solving method is the Cholesky decomposition described in Ch. 2.9, which is very fast but has some requirements for the matrix. Luckily these requirements are met by the reduced global stiffness matrix.

2.4 Shape Functions

Shape functions are the expressions that gives the ”allowed” ways the element can deform. There are some requirements that shape functions must fulfill for the stresses to converge towards the correct values (Bell, 2013), these are:

- *Continuity* - The field variables and their derivatives must be continuous up to and including order m-1, where m is the order of differentiation in the strain-displacement relation.
- *Completeness* - textbfNu (displacement field times displacement vector) must be able to represent rigid body movement without producing stresses in the element, and for certain dof values produce a state of constant stress.
- *The interpolation requirement* - The first requirement is valid for all elements, while the second only for 2D and 3D cases, and the third only for displacement dofs.
  1. The shape function N_i must yield $u_i = 1$, while $u_j = 0$ where $(j \neq i)$.
  2. $N_i = 0$ for all sides and surfaces which does not contain dof i.
  3. $\sum N_i = 1$, the sum of all shape function must be one.

2.5 Beam Elements

2.5.1 Beam Element Shape Functions

Although solving the global stiffness matrix for the applied loads, the resulting values only give information about displacements at supplied nodes. For information about how the displacements look *inside* each element, the (sub-element) displacements must be interpolated from the nodal displacements. A way of achieving this is by applying an assumed displacement field (Saouma, 1999). The displacement field is an assumed polynomial which aims to approximate the deformation shape of the element. Mathematically, this may be expressed as:

$$\mathbf{\Delta} = \sum_{i=1}^{n} N_i(x)\Delta_{l,i}^{e} = \mathbf{N}(\mathbf{x})\mathbf{\Delta_l^e} \qquad \text{(Eq. 2.5.1)}$$

where

1. $\mathbf{\Delta_l}$ = local generalized displacement
2. $\mathbf{\Delta_l^e}$ = element's local nodal displacement
3. $N(x)_i$ = shape functions
4. $\mathbf{N(x)}$ = displacement field

$\mathbf{\Delta_l^e}$ is defined as:

$$\mathbf{\Delta_l^e} = \begin{bmatrix} u_{x,1} & u_{y,1} & u_{z,1} & \theta_{x,1} & \theta_{y,1} & \theta_{z,1} & u_{x,2} & u_{y,2} & u_{z,2} & \theta_{x,2} & \theta_{y,2} & \theta_{z,2} \end{bmatrix}^T$$

The number of shape functions are dependent on the number of dofs, as well as desired continuity. Continuity pertains to the reproduction of deflection and curvature. The degree of continuity decides whether the displacements are constant or requires continuity of slopes.

Note that the nodal displacement vector for element e, $\mathbf{\Delta^e}$, which is calculated by Cholesky Banachiewicz as shown in Ch. 2.9 must be transformed from global to local coordinates before multiplied with the displacement field. The transformation matrix is a 12x12 matrix like the ones from Eq. 2.7.27-2.7.29.

$$\mathbf{\Delta_l^e} = T\mathbf{\Delta^e} \qquad \text{(Eq. 2.5.2)}$$

After calculating the generalized deformations by Eq. 2.5.1, the resulting displacements for each new sub element, such as u_x, u_y, u_z, θ_x, θ_y and θ_z, must then be transformed back to global coordinates using the following equation

$$\mathbf{\Delta} = T^T \mathbf{\Delta_1} \tag{Eq. 2.5.3}$$

Axial and Torsional Shape Function

Both axial force and torsion is constant along the length of the element (since St. Venant's Torsion is assumed). This means that both axial and torsional displacements are linear and can be approximated using the same shape function. Deriving these shape functions are done by starting from the linear polynomial

$$u = ax + b \tag{Eq. 2.5.4}$$

Coefficients can be found by applying boundary conditions

$$u(x = 0) = u_1 = 0 + b = b \tag{Eq. 2.5.5}$$

$$u(x = L) = u_2 = aL + b = aL + u_1 \tag{Eq. 2.5.6}$$

L is the element's local length along the X-axis. Solving for a and b yields

$$a = \frac{u_2 - u_1}{L} = \frac{u_2}{L} - \frac{u_1}{L} \qquad\qquad b = u_1 \tag{Eq. 2.5.7}$$

Substituting Eq. 2.5.7 into Eq. 2.5.4 gives

$$\begin{aligned}
u &= ax + b \\
&= (\frac{u_2}{L} - \frac{u_1}{L})x + u_1 && \text{(Eq. 2.5.8)} \\
&= \frac{u_2}{L}x - \frac{u_1}{L}x + u_1 && \text{(Eq. 2.5.9)} \\
&= (1 - \frac{x}{L})u_1 + \frac{x}{L}u_2 && \text{(Eq. 2.5.10)} \\
&= N_1 u_1 + N_2 u_2 && \text{(Eq. 2.5.11)}
\end{aligned}$$

The shape functions for axial and torsional displacement are then defined as

$$N_1 = 1 - \frac{x}{L} \qquad\qquad N_2 = \frac{x}{L} \tag{Eq. 2.5.12}$$

With this there are shape functions representing two out of six dofs.

The procedure can be sped up by use of matrix notation. Going back to Eq. 2.5.4, $u(x)$ can be described by the polynomial vector $\mathbf{p}$ and the coefficient vector $\boldsymbol{\Psi}$

$$u = ax + b = \begin{bmatrix} x & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \mathbf{p}\boldsymbol{\Psi} \qquad \text{(Eq. 2.5.13)}$$

Multiplying $\boldsymbol{\Psi}$ with the boundary condition matrix $\boldsymbol{\Upsilon}$ constructed from Eq. 2.5.5-2.5.6 gives the displacements

$$\boldsymbol{\Delta}_{\mathbf{a}} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ L & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \boldsymbol{\Upsilon}\boldsymbol{\Psi} \qquad \text{(Eq. 2.5.14)}$$

Here $\boldsymbol{\Delta}_{\mathbf{a}}$ are the nodal axial and torsional parts of $\boldsymbol{\Delta}^{\mathbf{e}}_{\mathbf{l}}$. The exact same procedure can be done for the rotational parts $\boldsymbol{\Delta}_{\mathbf{r}}$.

By inverting $\boldsymbol{\Upsilon}$, $\boldsymbol{\Psi}$ is now defined from $\boldsymbol{\Upsilon}$ and $\mathbf{u}$

$$\boldsymbol{\Delta}_{\mathbf{a}} = \boldsymbol{\Upsilon}\boldsymbol{\Psi} \implies \boldsymbol{\Upsilon}^{-\mathbf{1}}\boldsymbol{\Delta}_{\mathbf{a}} = \boldsymbol{\Upsilon}^{-\mathbf{1}}\boldsymbol{\Upsilon}\boldsymbol{\Psi} = \boldsymbol{\Psi} \qquad \text{(Eq. 2.5.15)}$$

Inversion of $\boldsymbol{\Upsilon}$

$$\boldsymbol{\Upsilon}^{-\mathbf{1}} = \frac{1}{det(\boldsymbol{\Upsilon})} \begin{bmatrix} 1 & -1 \\ -L & 0 \end{bmatrix} = \frac{1}{-L} \begin{bmatrix} 1 & -1 \\ -L & 0 \end{bmatrix} = \frac{1}{L} \begin{bmatrix} -1 & 1 \\ L & 0 \end{bmatrix} \qquad \text{(Eq. 2.5.16)}$$

The coefficient values are thus given as

$$\boldsymbol{\Psi} = \boldsymbol{\Upsilon}^{-\mathbf{1}}\boldsymbol{\Delta}_{\mathbf{a}} = \frac{1}{L} \begin{bmatrix} -1 & 1 \\ L & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \qquad \text{(Eq. 2.5.17)}$$

By multiplying the polynomials $\mathbf{p}$ with the coefficients $\boldsymbol{\Psi}$ calculated from Eq. 2.5.17, the interpolated displacements $\mathbf{u}$ can be found. Substituting Eq. 2.5.17 into Eq. 2.5.14 gives

$$\mathbf{u} = \mathbf{p}\boldsymbol{\Psi} = \mathbf{p}\boldsymbol{\Upsilon}^{-\mathbf{1}}\boldsymbol{\Delta}_{\mathbf{a}} \qquad \text{(Eq. 2.5.18)}$$

$$= \begin{bmatrix} x & 1 \end{bmatrix} \frac{1}{L} \begin{bmatrix} -1 & 1 \\ L & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} (1 - \frac{x}{L}) & \frac{x}{L} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \mathbf{N}\boldsymbol{\Delta}_{\mathbf{a}} \qquad \text{(Eq. 2.5.19)}$$

As can be observed, **N** can quickly be found by solving

$$\mathbf{N} = \mathbf{p}\boldsymbol{\Upsilon}^{-1} \qquad \text{(Eq. 2.5.20)}$$

Eq. 2.5.20 can be used to easily derive shape functions for flexural dofs as well.

Flexural Shape Functions

Since axial and torsional dofs now are in place, the remaining displacements are u_y, u_z, θ_y and θ_z. four more dofs need their associated shape functions, hence four more shape functions need to be found. Four boundary conditions are used to find these shape functions, meaning that the polynomial must be of order three (Saouma, 1999), the polynomial is assumed as follows for displacements

$$u = ax^3 + bx^2 + cx + d = \begin{bmatrix} x^3 & x^2 & x & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \mathbf{p}\boldsymbol{\Psi} \qquad \text{(Eq. 2.5.21)}$$

where the rotational displacements are defined as

$$\theta = \frac{du}{dx} = 3ax^2 + 2bx + c \qquad \text{(Eq. 2.5.22)}$$

Applying boundary conditions gives

$$u(x=0) = u_1 \qquad \left.\frac{du}{dx}\right|_{x=0} = \theta_1 \qquad \text{(Eq. 2.5.23)}$$

$$u(x=L) = u_2 \qquad \left.\frac{du}{dx}\right|_{x=L} = \theta_2 \qquad \text{(Eq. 2.5.24)}$$

Converting Eq. 2.5.21 to matrix notation using the notation from Eq. 2.5.23-2.5.24 yields the following boundary condition matrix $\boldsymbol{\Upsilon}$

$$\boldsymbol{\Delta}_\mathbf{f} = \begin{bmatrix} u_1 \\ \theta_1 \\ u_2 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ L^3 & L^2 & L & 1 \\ 3L^2 & 2L & 1 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \boldsymbol{\Upsilon}\boldsymbol{\Psi} \qquad \text{(Eq. 2.5.25)}$$

Here $\mathbf{\Delta_f}$ is the nodal flexural part of $\mathbf{\Delta_l^e}$. The inverted $\mathbf{\Upsilon}$-matrix is

$$\mathbf{\Upsilon^{-1}} = \frac{1}{L^3}\begin{bmatrix} 2 & L & -2 & L \\ -3L & -2L^2 & 3L & -L^2 \\ 0 & L^3 & 0 & 0 \\ L^3 & 0 & 0 & 0 \end{bmatrix} \quad \text{(Eq. 2.5.26)}$$

By use of Eq. 2.5.20, **N** is found to be

$$\mathbf{N} = \mathbf{p}\mathbf{\Upsilon^{-1}} = \begin{bmatrix} x^3 & x^2 & x & 1 \end{bmatrix} \frac{1}{L^3}\begin{bmatrix} 2 & L & -2 & L \\ -3L & -2L^2 & 3L & -L^2 \\ 0 & L^3 & 0 & 0 \\ L^3 & 0 & 0 & 0 \end{bmatrix} \quad \text{(Eq. 2.5.27)}$$

$$= \begin{bmatrix} (\frac{2x^3}{L^3} - \frac{3x^2}{L^2} + 1) & (x - \frac{2x^2}{L} + \frac{x^3}{L^2}) & (\frac{3x^2}{L^2} - \frac{2x^3}{L^3}) & (\frac{x^3}{L^2} - \frac{x^2}{L}) \end{bmatrix} \quad \text{(Eq. 2.5.28)}$$

The Complete Shape Functions

Now all shape functions are found, representing all six dofs (in each node):

$$N_1 = 1 - \frac{x}{L} \quad \text{(Eq. 2.5.29)}$$

$$N_2 = \frac{x}{L} \quad \text{(Eq. 2.5.30)}$$

$$N_3 = 1 - 3\frac{x^2}{L^2} + 2\frac{x^3}{L^3} \quad \text{(Eq. 2.5.31)}$$

$$N_4 = x - 2\frac{x^2}{L} + \frac{x^3}{L^2} \quad \text{(Eq. 2.5.32)}$$

$$N_5 = 3\frac{x^2}{L^2} - 2\frac{x^3}{L^3} \quad \text{(Eq. 2.5.33)}$$

$$N_6 = \frac{x^3}{L^2} - \frac{x^2}{L} \quad \text{(Eq. 2.5.34)}$$

Notice that several shape functions are almost identical, meaning shortcuts can be made to avoid recalculating them in the program. Addition and subtraction operations have shorter time execution costs than division and exponentiation. Some simplification can be made as

$$N_1 = 1 - N_2 \qquad\qquad N_5 = -N_3 + 1$$

The first order derived shape functions can be useful for finding strains, stresses and internal forces related to the axial and torsional deformations. Deriving the shape functions yields the following equations

$$dN_1 = -\frac{1}{L} \tag{Eq. 2.5.35}$$

$$dN_2 = \frac{1}{L} \tag{Eq. 2.5.36}$$

$$dN_3 = -6\frac{x}{L^2} + 6\frac{x^2}{L^3} \tag{Eq. 2.5.37}$$

$$dN_4 = 1 - 4\frac{x}{L} + 3\frac{x^2}{L^2} \tag{Eq. 2.5.38}$$

$$dN_5 = 6\frac{x}{L^2} - 6\frac{x^2}{L^3} \tag{Eq. 2.5.39}$$

$$dN_6 = 3\frac{x^2}{L^2} - 2\frac{x}{L} \tag{Eq. 2.5.40}$$

Similarly to N_2 and N_4, dN_2 and dN_5 can freed from recalculation.

$$dN_2 = -dN_1 \qquad\qquad dN_5 = -dN_3$$

Remember that θ_y and θ_z are defined as

$$\theta_y = \frac{du_z}{dx} \qquad\qquad \theta_z = \frac{du_y}{dx} \tag{Eq. 2.5.41}$$

This means that θ_y and θ_z are calculated from the derived displacement field that will be presented later.

The second order derivative shape functions are useful for finding strains and stresses, but follows the same logic as before so are not shown.

Displacement Fields

The shape functions are used to construct a *displacement field* **N** which is used to approximate a displacement pattern, as per Eq. 2.5.1. When multiplied by the nodal displacements per element, $\mathbf{\Delta_I^e}$, the displacement field **N** represents the general deformations of u_x, u_y, u_z and θ_x. The first order derived displacement field **dN** can be used to find θ_y and θ_z, see Eq. 2.5.41.

$$\mathbf{N} = \begin{array}{c} \begin{array}{cccccccccccc} u_{x,1} & u_{y,1} & u_{z,1} & \theta_{x,1} & \theta_{y,1} & \theta_{z,1} & u_{x,2} & u_{y,2} & u_{z,2} & \theta_{x,2} & \theta_{y,2} & \theta_{z,2} \end{array} \\ \begin{bmatrix} N_1 & 0 & 0 & 0 & 0 & 0 & N_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & N_3 & 0 & 0 & 0 & N_4 & 0 & N_5 & 0 & 0 & 0 & N_6 \\ 0 & 0 & N_3 & 0 & -N_4 & 0 & 0 & 0 & N_5 & 0 & -N_6 & 0 \\ 0 & 0 & 0 & N_1 & 0 & 0 & 0 & 0 & 0 & N_2 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} \\ u_x \\ u_y \\ u_z \\ \theta_x \end{array} \qquad \text{(Eq. 2.5.42)}$$

$$\mathbf{dN} = \frac{d\mathbf{N}}{dx} = \begin{array}{c} \begin{array}{cccccccccccc} u_{x,1} & u_{y,1} & u_{z,1} & \theta_{x,1} & \theta_{y,1} & \theta_{z,1} & u_{x,2} & u_{y,2} & u_{z,2} & \theta_{x,2} & \theta_{y,2} & \theta_{z,2} \end{array} \\ \begin{bmatrix} dN_1 & 0 & 0 & 0 & 0 & 0 & dN_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & dN_3 & 0 & 0 & 0 & dN_4 & 0 & dN_5 & 0 & 0 & 0 & dN_6 \\ 0 & 0 & dN_3 & 0 & -dN_4 & 0 & 0 & 0 & N_5 & 0 & -dN_6 & 0 \\ 0 & 0 & 0 & dN_1 & 0 & 0 & 0 & 0 & 0 & dN_2 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} \\ \frac{du_x}{dx} \\ \frac{du_y}{dx} = \theta_z \\ \frac{du_z}{dx} = \theta_y \\ \frac{d\theta_x}{dx} \end{array} \qquad \text{(Eq. 2.5.43)}$$

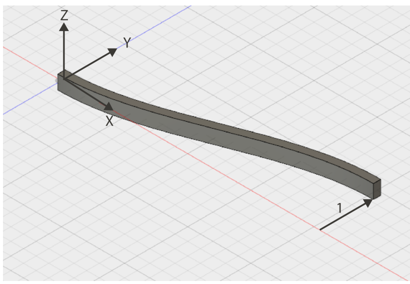
If there is a nodal displacement of 1 in the Y-direction ($u_{y,2} = 1$), as in Fig 2.3a, the (transposed) displacement vector $\mathbf{\Delta_I^e}$ will look like

$$\mathbf{\Delta_I^e} = \begin{array}{c} \begin{array}{cccccccccccc} u_{x1} & u_{y1} & u_{z1} & \theta_{x1} & \theta_{y1} & \theta_{z1} & u_{x2} & u_{y2} & u_{z2} & \theta_{x2} & \theta_{y2} & \theta_{z2} \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}^T \qquad \text{(Eq. 2.5.44)}$$

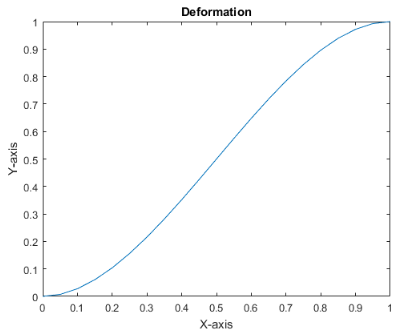
By using Eq. 2.5.1 on both **N** and **dN**, then retrieving appropriate values, the displacement vector becomes

$$\mathbf{\Delta_1} = \begin{bmatrix} u_x \\ u_y \\ u_z \\ \theta_x \\ \theta_y \\ \theta_z \end{bmatrix} = \begin{bmatrix} 0 \\ N_5 \\ 0 \\ 0 \\ 0 \\ dN_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 3\frac{x^2}{L^2} - 2\frac{x^3}{L^3} \\ 0 \\ 0 \\ 0 \\ 6\frac{x}{L^2} - 6\frac{x^2}{L^3} \end{bmatrix} \qquad \text{(Eq. 2.5.45)}$$

Assuming $L = 1$ and incrementing values for x at intervals of 0.05, the resulting u_y displacement looks much like expected, see Fig 2.3b



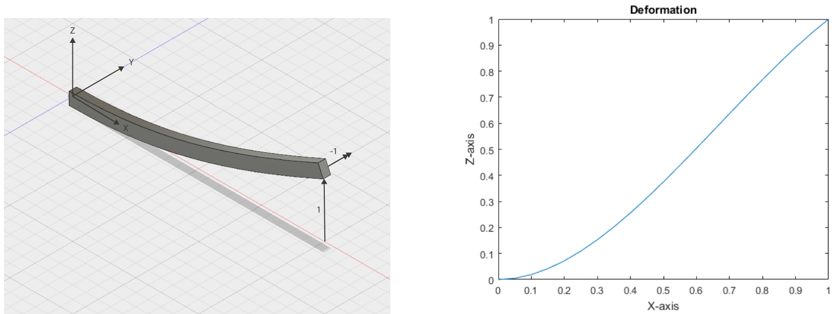
(a) Assumed deformation



(b) Interpolated deformation

Figure 2.3: Nodal displacement of 1 in Y-direction

An example following the same procedure for a displacement situation like on Fig. 2.4a, where $u_{z,2} = 1$ and $\theta_y = -1$ results in a displacement pattern like on Fig. 2.4b. Notice that this case, where $\theta_y = -1$, illustrates why $\mathbf{N_{3,5}}$ and $\mathbf{N_{3,11}}$ are negative in the displacement matrix since rotation about the Y-axis contributes negatively to the u_z value.



(a) Assumed deformation (b) Interpolated deformation

Figure 2.4: Nodal displacement of 1 in Z-direction and -1 about the Y-axis

2.5.2 One-Dimensional Stress and Strain

For trusses there is only one type of stress, namely axial stress σ_x. Since there are no moments, stress is defined as

$$\sigma = \sigma_x = \frac{F}{A} \qquad \text{(Eq. 2.5.46)}$$

According to Hooke's Law, the relation between stress and strain for a linearly elastic material is

$$\sigma = E\varepsilon \qquad \text{(Eq. 2.5.47)}$$

Since there is only axial stress in trusses, there also only be axial strain ε_x. The definition of strain along an element is

$$\varepsilon_x = \frac{\partial u}{\partial x} = \frac{u_2 - u_1}{L} \qquad \text{(Eq. 2.5.48)}$$

Here u_1 and u_2 are the length displacements of respectively node 1 and 2. By reformulating u from Eq. 2.5.11 we end up with

$$u(x) = N_1(x)u_1 + N_2(x)u_2 = u_1 + \frac{u_2 - u_1}{L}x \qquad \text{(Eq. 2.5.49)}$$

As can be observed, $\varepsilon_x = \frac{du}{dx}$, which means that the axial strain can be found by calculating the displacement field **dN** from Eq. 2.5.43 and multiplying with the nodal displacement vector $\mathbf{\Delta_a^e}$.

$$\varepsilon_{x,axial} = \frac{du_x}{dx} = \frac{d\mathbf{N_1}}{dx}\mathbf{\Delta_a^e} \qquad \text{(Eq. 2.5.50)}$$

Alternatively, the change in length can be calculated manually from the displacement in x-, y-, and z-direction in both nodes.

For beams, there are two common models for straight, prismatic beams, namely Euler-Bernoulli beam theory and Timoshenko beam theory (Aalberg, 2014). The main difference between the two lies in their premises. Euler-Bernoulli does not include shear deformations, which means that cross-sections remain normal to the neutral axis after deformation. Timoshenko includes shear deformations, meaning that there may be rotation between the cross-section and the bending line, as well as stresses in other than the length direction. In this thesis, Euler-Bernoulli beam theory has been used.

While more thoroughly explained in Ch. 2.6.3 on plate bending, since rotations are very small, let us assume that

$$u = z\theta_y \qquad \text{where} \qquad \theta_y = \frac{du_z}{dx} = \frac{d\mathbf{N_3}}{dx}\mathbf{\Delta^e} \tag{Eq. 2.5.51}$$

Bending strain from rotation about the y-axis is given by

$$\varepsilon_{xx,y} = \frac{\partial u}{\partial x} = z\frac{d\theta_y}{dx} = z\frac{d^2u_z}{dx^2} = z\frac{d^2\mathbf{N_3}}{dx^2}\mathbf{\Delta_l^e} \tag{Eq. 2.5.52}$$

Here $\mathbf{N_3}$ is the third row of Eq. 2.5.42. Bending about the Z-axis bring about a negative contribution, which means that biaxial bending can be written as

$$\varepsilon_{xx,z} = -y\frac{d\theta_z(x)}{dx} \implies \varepsilon_{xx,bending} = z\frac{d\theta_y(x)}{dx} - y\frac{d\theta_z(x)}{dx} \tag{Eq. 2.5.53}$$

Combining the axial contribution from Eq. 2.5.50 and bending contribution from Eq. 2.5.54 to the internal strain energy, ε_{xx} is defined as

$$\varepsilon_{xx} = \frac{du_x}{dx} + z\frac{d^2u_z}{dx^2} - y\frac{d^2u_y}{dx^2} \tag{Eq. 2.5.54}$$

The maximum strain energy $\varepsilon_{xx,max}$ is useful and can be found by taking the absolute values while respecting the polarity of the axial strain. This means that positive axial strain (elongated element) will result in a positive ε_{xx}, while a negative axial strain will result in a negative ε_{xx}.

$$\frac{d\mathbf{N_1}}{dx}\mathbf{\Delta_l^e} > 0 \implies \varepsilon_{xx} = \frac{d\mathbf{N_1}}{dx}\mathbf{\Delta_l^e} + |z\frac{d^2\mathbf{N_3}}{dx^2}\mathbf{\Delta_l^e}| + |y\frac{d^2\mathbf{N_2}}{dx^2}\mathbf{\Delta_l^e}| \tag{Eq. 2.5.55}$$

$$\frac{d\mathbf{N_1}}{dx}\mathbf{\Delta_l^e} <= 0 \implies \varepsilon_{xx} = \frac{d\mathbf{N_1}}{dx}\mathbf{\Delta_l^e} - |z\frac{d^2\mathbf{N_3}}{dx^2}\mathbf{\Delta_l^e}| - |y\frac{d^2\mathbf{N_2}}{dx^2}\mathbf{\Delta_l^e}| \tag{Eq. 2.5.56}$$

2.5.3 Element Stiffness Matrix

The beam element stiffness matrix can be derived through the shape functions found in Ch. 2.5.1 because the chosen flexural shape function happens to be the exact solution for the partial differential equation of Euler-Bernoulli beam theory. Through use of the principal of virtual displacement an expression for the element stiffness matrix can be established (Bell, 2013). In the case of axial stress (Saouma, 1999), as in a truss, it becomes

$$\mathbf{k}^e_{axial} = \int_{V_e} \mathbf{B}^T_{axial} E \mathbf{B}_{axial} dv \qquad \text{(Eq. 2.5.57)}$$

The $\mathbf{B}_{axial}$ matrix for an axial stress case can be extracted from Eq. 2.5.43 in combination with Eq. 2.5.50 as

$$\mathbf{B}_{axial} = \begin{bmatrix} \frac{dN_1}{dx} & \frac{dN_2}{dx} \end{bmatrix} = \begin{bmatrix} -\frac{1}{L} & \frac{1}{L} \end{bmatrix} \qquad \text{(Eq. 2.5.58)}$$

Which when all terms are constant gives

$$\mathbf{k}^e_{axial} = A \int_0^L \begin{bmatrix} -\frac{1}{L} \\ \frac{1}{L} \end{bmatrix} E \begin{bmatrix} -\frac{1}{L} & \frac{1}{L} \end{bmatrix} dx = \frac{AE}{L} \begin{array}{c} \begin{array}{cc} u_{x,1} & u_{x,2} \end{array} \\ \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \end{array} \begin{array}{c} F_{x1} \\ F_{x2} \end{array} \qquad \text{(Eq. 2.5.59)}$$

This can also be utilized to represent the axial forces in a beam element, as shall be shown later. As mentioned in Ch. 2.5.1 the axial and torsional parts share the same shape functions and thus the torsional part can be shown to be

$$\mathbf{k}^e_{torsion} = \frac{GJ}{L} \begin{array}{c} \begin{array}{cc} \theta_{x,1} & \theta_{x,2} \end{array} \\ \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \end{array} \begin{array}{c} T_{x,1} \\ T_{x,2} \end{array} \qquad \text{(Eq. 2.5.60)}$$

Where J is the torsional constant which is dependent on the cross-sectional shape, and G for an isotropic material becomes

$$G = \frac{E}{2(1+\nu)} \qquad \text{(Eq. 2.5.61)}$$

For a flexural element the expression for the stiffness matrix, quite similar to Eq. 2.5.57 with a few extra terms from Eq. 2.5.52, becomes

$$\mathbf{k}^e_{flex} = \int_0^L \int_{A_e} \mathbf{B}^T_{flex} E \mathbf{B}_{flex} z^2 dA dx \tag{Eq. 2.5.62}$$

Where the **B** matrix for a flexural element can be found from Eq. 2.5.37 - 2.5.40 as

$$\mathbf{B}_{flex} = \begin{bmatrix} (-6\frac{x}{L^2} + 6\frac{x^2}{L^3}) & (1 - 4\frac{x}{L} + 3) & (\frac{x}{L^2} - 6\frac{x^2}{L^3}) & (\frac{x^2}{L^2} - 2\frac{x}{L}) \end{bmatrix} \tag{Eq. 2.5.63}$$

And knowing that

$$\int_{A_e} z^2 dA = I_y \tag{Eq. 2.5.64}$$

Which gives the flexural stiffness matrix expression as

$$\mathbf{k}^e_{flex} = EI_y \int_0^L \mathbf{B}^T \mathbf{B} dx \tag{Eq. 2.5.65}$$

Thus, the flexural element stiffness matrix can be written as

$$\mathbf{k}^e_{flex} = \frac{EI}{L^3} \begin{array}{c} \begin{array}{cccc} u_{z,1} & \theta_{y,1} & u_{z,2} & \theta_{y,2} \end{array} \\ \begin{bmatrix} 12 & -6L & -12 & -6L \\ -6L & 4L^2 & 6L & 2L^2 \\ -12 & 6L & 12 & 6L \\ -6L & 2L^2 & 6L & 4L^2 \end{bmatrix} \end{array} \begin{array}{c} \\ V_{z1} \\ M_{y1} \\ V_{z2} \\ M_{y2} \end{array} \tag{Eq. 2.5.66}$$

The same procedure can be repeated to find $u_{y1}, \theta_{z1}, u_{y2}, \theta_{z2},$.

The stiffness matrix for axial, torsional and flexural deformations can now be assembled into an element stiffness matrix for a beam element, also called 3D frame element. The assembly will result in

$$\mathbf{k}^{\mathbf{e}}_{\mathbf{beam}} = \frac{E}{L^3} \begin{array}{cc} \begin{matrix} u_{x1} & u_{y1} & u_{z1} & \theta_{x1} & \theta_{y1} & \theta_{z1} & u_{x2} & u_{y2} & u_{z2} & \theta_{x2} & \theta_{y2} & \theta_{z2} \end{matrix} & \\ \begin{bmatrix} AL^2 & 0 & 0 & 0 & 0 & 0 & -AL^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 12I_z & 0 & 0 & 0 & 6I_zL & 0 & -12I_z & 0 & 0 & 0 & 6I_zL \\ 0 & 0 & 12I_y & 0 & -6I_yL & 0 & 0 & 0 & -12I_y & 0 & -6I_yL & 0 \\ 0 & 0 & 0 & \frac{I_xL^2}{2(1+\nu)} & 0 & 0 & 0 & 0 & 0 & -\frac{I_xL^2}{2(1+\nu)} & 0 & 0 \\ 0 & 0 & -6I_yL & 0 & 4I_yL^2 & 0 & 0 & 0 & 6I_yL & 0 & 2I_yL^2 & 0 \\ 0 & 6I_zL & 0 & 0 & 0 & 4I_zL^2 & 0 & -6I_zL & 0 & 0 & 0 & 2I_zL^2 \\ -AL^2 & 0 & 0 & 0 & 0 & 0 & AL^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -12I_z & 0 & 0 & 0 & -6I_zL & 0 & 12I_z & 0 & 0 & 0 & -6I_zL \\ 0 & 0 & -12I_y & 0 & 6I_yL & 0 & 0 & 0 & 12I_y & 0 & 6I_yL & 0 \\ 0 & 0 & 0 & -\frac{I_xL^2}{2(1+\nu)} & 0 & 0 & 0 & 0 & 0 & \frac{I_xL^2}{2(1+\nu)} & 0 & 0 \\ 0 & 0 & -6I_yL & 0 & 2I_yL^2 & 0 & 0 & 0 & 6I_yL & 0 & 4I_yL^2 & 0 \\ 0 & 6I_zL & 0 & 0 & 0 & 2I_zL^2 & 0 & -6I_zL & 0 & 0 & 0 & 4I_zL^2 \end{bmatrix} & \begin{matrix} F_{x1} \\ V_{y1} \\ V_{z1} \\ T_{x1} \\ M_{y1} \\ M_{z1} \\ F_{x2} \\ V_{y2} \\ V_{z2} \\ T_{x2} \\ M_{y2} \\ M_{z2} \end{matrix} \end{array}$$

(Eq. 2.5.67)

2.6 Triangular Shell Elements

There will only be focused at the triangular elements, this is because of their simplicity, versatility and robustness in usage and calculations (Bell, 2013). Many of the principles presented however can be utilized to derive the necessary equations for higher order closed polygon elements.

The triangular shell element is a plane 2D element, in contrast to the 1D truss or 3D solid elements. To achieve adequate results with the 2D triangular element, the structural problem should be a thin plate/shell structure. For a shell or plate to be considered thin, it must have a thickness less than approximately 1/10 of the span length (Mike A., 2016). A thickness less than this is very often the case when plates and shells are used, which is why the 2D plane element has been chosen.

It should also be noted that the reason for it to be adequate with 2D elements for thin shell is because the shear deformation out of plane is negligible compared to the bending deformation. This implies that for medium thick, thick plates and thick shells, 3D solid elements that takes shear deformation into account, is considerably better. The 3D solid elements are in general more accurate but also more demanding in terms of computing power. Because of the increased amount of dofs, they are also more time consuming. From a parametric real-time calculation perspective, the 2D plane elements has sufficient accuracy with respect to time utilization.

The triangular shell element can be said to consist of two main parts, the in-plane stresses and strains, also called membrane part, and the bending part. These parts can be viewed as completely separate, if some assumptions are made, and can therefore be formulated and calculated before they are assembled together in the element stiffness matrix and solved for deformations.

Isotropic material for the shell element is assumed during the derivations that follows. When a material is isotropic it means that the material properties is equal in all directions, that is

$$E_x = E_y = E_z = E \qquad \text{(Eq. 2.6.1)}$$

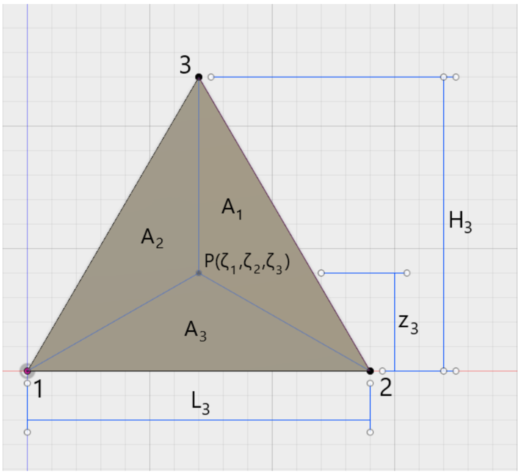
in contrast to orthotropic which is a type of orthogonal anisotropy where

$$E_x \neq E_y \neq E_z \qquad \text{(Eq. 2.6.2)}$$

2.6.1 Area Coordinates

To streamline the derivation of the triangular element stiffness matrix it is often advantageous to use area coordinates to derive the necessary relations. The area coordinate i is a normalized distance to edge i, so the area coordinates can be defined from Fig. 2.5 as

$$\zeta_i = \frac{A_i}{A} = \frac{\frac{1}{2}z_i L_i}{\frac{1}{2}H_i L_i} = \frac{z_i}{H_i} \qquad \text{(Eq. 2.6.3)}$$



Since

$$A = \sum_i^3 A_i \qquad \text{(Eq. 2.6.4)}$$

it can be stated that

$$\zeta_1 + \zeta_2 + \zeta_3 = 1 \qquad \text{(Eq. 2.6.5)}$$

Figure 2.5: Area coordinate relations

It should be specified that the numbering sequence must be in counter clockwise order for the following derivation to be applicable.

To use the area coordinates, a transformation between Cartesian coordinates and area coordinates are necessary to later fit the element into a global system. By inspection of Fig. 2.5 it can be seen that the area coordinate for point i increases towards 1 the closer it gets to node i. From this, the following transformation emerges

$$x = x_1\zeta_1 + x_2\zeta_2 + x_3\zeta_3 \qquad \text{(Eq. 2.6.6)}$$

$$y = y_1\zeta_1 + y_2\zeta_2 + y_3\zeta_3 \qquad \text{(Eq. 2.6.7)}$$

In an easily invertible matrix form this is

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \zeta_1 \\ \zeta_2 \\ \zeta_3 \end{bmatrix} \tag{Eq. 2.6.8}$$

It can also be shown that the determinant of this matrix is equal to twice the triangle area (Bell, 2013), much like in Eq. 2.6.61. The inverse of this then becomes

$$\begin{bmatrix} \zeta_1 \\ \zeta_2 \\ \zeta_3 \end{bmatrix} = \frac{1}{2A} \begin{bmatrix} y_{23} & x_{32} & (x_2y_3 - x_3y_2) \\ y_{31} & x_{13} & (x_3y_1 - x_1y_3) \\ y_{12} & x_{21} & (x_1y_2 - x_2y_1) \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{Eq. 2.6.9}$$

$$x_{ij} = x_i - x_j \qquad y_{ij} = y_i - y_j \qquad \text{and} \qquad A = \text{area of triangle}$$

From Eq. 2.6.6 and Eq. 2.6.7, the derivative relations are defined as

$$\frac{\partial x}{\partial \zeta_i} = x_i \quad \text{and} \quad \frac{\partial y}{\partial \zeta_i} = y_i \tag{Eq. 2.6.10}$$

By combining Eq. 2.6.10 and Eq. 2.6.9, it becomes

$$\frac{\partial \zeta_1}{\partial x} = \frac{y_{23}}{2A} \qquad \frac{\partial \zeta_2}{\partial x} = \frac{y_{31}}{2A} \qquad \frac{\partial \zeta_3}{\partial x} = \frac{y_{12}}{2A} \tag{Eq. 2.6.11}$$

$$\frac{\partial \zeta_1}{\partial y} = \frac{x_{32}}{2A} \qquad \frac{\partial \zeta_2}{\partial y} = \frac{x_{13}}{2A} \qquad \frac{\partial \zeta_3}{\partial y} = \frac{x_{21}}{2A} \tag{Eq. 2.6.12}$$

If an arbitrary function $f(\zeta_1, \zeta_2, \zeta_3)$ shall be derived, the above expressions can be assembled as

$$\frac{\partial f}{\partial x} = \frac{1}{2A}\left(\frac{\partial f}{\partial \zeta_1} y_{23} + \frac{\partial f}{\partial \zeta_2} y_{31} + \frac{\partial f}{\partial \zeta_3} y_{12}\right) \tag{Eq. 2.6.13}$$

$$\frac{\partial f}{\partial y} = \frac{1}{2A}\left(\frac{\partial f}{\partial \zeta_1} x_{32} + \frac{\partial f}{\partial \zeta_2} x_{13} + \frac{\partial f}{\partial \zeta_3} x_{21}\right) \tag{Eq. 2.6.14}$$

In matrix notation, this is

$$\begin{bmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{bmatrix} = \frac{1}{2A} \begin{bmatrix} y_{23} & y_{31} & y_{12} \\ x_{32} & x_{13} & x_{21} \end{bmatrix} \begin{bmatrix} \dfrac{\partial}{\partial \zeta_1} \\ \dfrac{\partial}{\partial \zeta_2} \\ \dfrac{\partial}{\partial \zeta_3} \end{bmatrix} \quad \text{(Eq. 2.6.15)}$$

Notice that there are three area coordinates for every two ”global” coordinates. This is easily fixed since the area coordinates are not independent, as seen from Eq. 2.6.5, and therefore

$$\zeta_3 = 1 - \zeta_1 - \zeta_2 \quad \text{(Eq. 2.6.16)}$$

There can now be established invertible and unambiguous expressions for differentiation where only the independent area coordinates are included. This can with the definition in Eq. 2.6.16 be written as

$$\begin{bmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{bmatrix} = \frac{1}{2A} \begin{bmatrix} y_{23} & y_{31} \\ x_{32} & x_{13} \end{bmatrix} \begin{bmatrix} \dfrac{\partial}{\partial \zeta_1} \\ \dfrac{\partial}{\partial \zeta_2} \end{bmatrix} \quad \text{(Eq. 2.6.17)}$$

The area coordinate derivation thus far is only valid for triangles with straight edges. With this, the area coordinate expressions for linear elements has been found.

2.6.2 Two-Dimensional Stress and Strain

In a plane element, the forces and deformations are simplified to be dependent on only two axes, namely x and y axis. The illustration on Fig. 2.6 shows the relevant stresses for a plate element. Notice that all stresses depending on the z axis has been neglected.

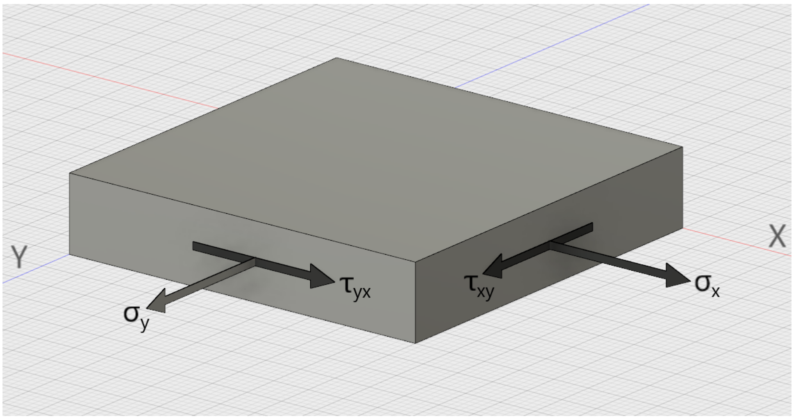


Figure 2.6: Two-dimensional stresses in plate element, equally on the opposite sides

From Eq. 2.5.47 an equation for the strain in each of these axes can be derived, but first it must be rearranged to solve for the strain in x direction.

$$\sigma_x = E_x \varepsilon_x \implies \varepsilon_x = \frac{\sigma_x}{E_x} \qquad \text{(Eq. 2.6.18)}$$

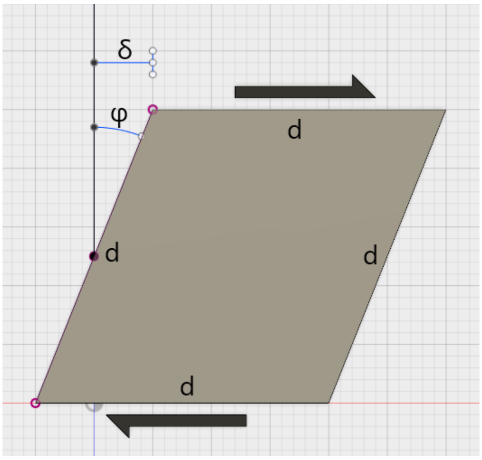
For a plane element, a strain in one direction will result in some strain in the other direction. This can be shown in a uniaxial stress test (Bell, 2013). This effect is called the *Poisson* effect. The general way of implementing this into the formulas is through *Poisson's ratio*, which is defined as

$$\nu_x = -\frac{\varepsilon_x}{\varepsilon_y} \qquad \text{(Eq. 2.6.19)}$$

Thus from Eq. 2.6.18 and Eq. 2.6.19, the two-dimensional strain in x direction is defined as

$$\varepsilon_x = \frac{\sigma_x}{E_x} - \nu_x \frac{\sigma_y}{E_y} \qquad \text{(Eq. 2.6.20)}$$

The same can be shown for strain in the y direction. Shear strain can be thought of in a similar manner since it relates to the rotational deformation, as illustrated in Fig. 2.7. For very small rotations the tangent of the angle can be approximated equal to the angle, and the shear strain can therefore be found as follows



$$\gamma_{xy} = 2\phi \qquad \text{(Eq. 2.6.21)}$$

where

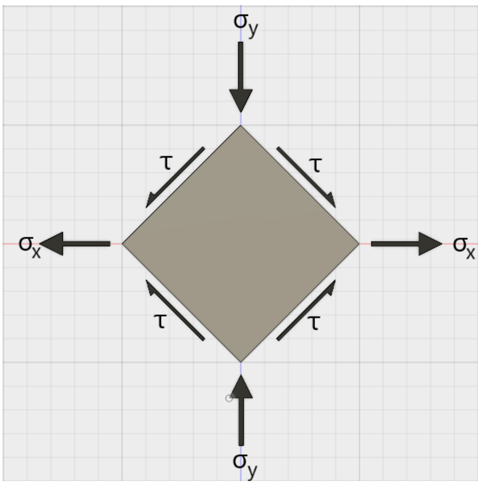
$$\phi \approx tan(\phi) = \frac{2\delta}{d} \qquad \text{(Eq. 2.6.22)}$$

which gives

$$\varepsilon_x = \frac{2\delta}{d} = \phi = \frac{\gamma_{xy}}{2} \qquad \text{(Eq. 2.6.23)}$$

Figure 2.7: Rotation from shear deformation

Whereas for an element like on Fig. 2.8, where main axes coincides with x and y axes, equilibrium requires that



$$\sigma = \sigma_x = \sigma_y \qquad \text{(Eq. 2.6.24)}$$

$$\tau = \tau_{xy} = \sigma \qquad \text{(Eq. 2.6.25)}$$

Figure 2.8: Axial and shear stress relation

If an isotropic material is assumed in Fig. 2.8, then Eq. 2.6.20 for the strain simplifies to

$$\varepsilon_x = \frac{1}{E}(\sigma - \nu(-\sigma)) = \frac{\sigma}{E}(1 + \nu) \qquad \text{(Eq. 2.6.26)}$$

Hence from Eq. 2.6.23

$$\frac{\gamma_{xy}}{2} = \varepsilon_x = \frac{\sigma}{E}(1+\nu) \tag{Eq. 2.6.27}$$

Substitution from Eq. 2.6.25 gives

$$\tau_{xy} = \frac{E}{2(1+\nu)}\gamma_{xy} = G\gamma_{xy} \tag{Eq. 2.6.28}$$

Here the G is called the *shear modulus*.

Moving on to matrix notation, the system for the strain-stress relation can be assembled from Eq. 2.6.20 and Eq. 2.6.28 as

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{bmatrix} = \frac{1}{E} \begin{bmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & 2(1+\nu) \end{bmatrix} \begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \mathbf{C^{-1}}\boldsymbol{\sigma} \tag{Eq. 2.6.29}$$

Here the $\mathbf{C}^{-1}$ matrix is called the flexibility matrix, and the inverse relation is

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{bmatrix} = \mathbf{C}\boldsymbol{\varepsilon} \tag{Eq. 2.6.30}$$

The $\mathbf{C}$ matrix is called the *elasticity matrix* and will be particularly important to the development of the general shell element. If the material is orthotropic, which means the stiffness is different for x and y direction, and we assume that Eq. 2.6.28 still is valid, it is clear from Eq. 2.6.20 that the strain becomes

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \end{bmatrix} = \begin{bmatrix} \frac{1}{E_x} & \frac{-\nu_x}{E_y} \\ \frac{-\nu_y}{E_x} & \frac{1}{E_x} \end{bmatrix} \begin{bmatrix} \sigma_x \\ \sigma_y \end{bmatrix} \tag{Eq. 2.6.31}$$

Which can be inverted to achieve the orthotropic equivalent to $\mathbf{C}$ from Eq. 2.6.29. It should be noted that the strain in z direction is regularly not zero as

$$\varepsilon_z = \frac{-\nu(\sigma_x + \sigma_y)}{E} \tag{Eq. 2.6.32}$$

However, this is of little consequence as it is a result of the lateral contraction from x and/or y direction, and σ_z is therefore still zero.

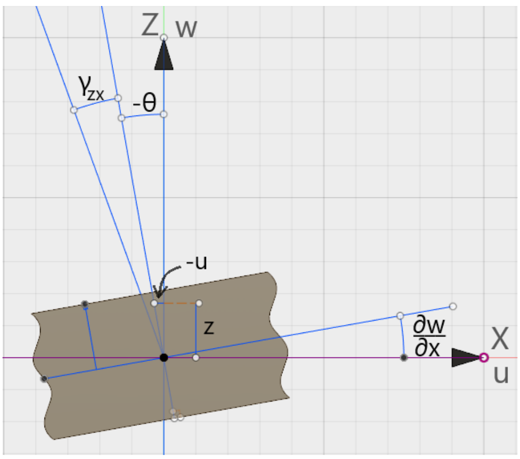
2.6.3 Plate Bending

For plate bending it is assumed that the element qualifies as a thin plate as described in Chapter 2.6.

The first additional assumption is that straight lines in the plate which is normal to the mid-surface, remains both straight and normal to the mid-surface after deformation, and the thickness remains after deformation. This means that strains varies linearly with the thickness of the plate. This is often called Kirchhoff-Love plate theory and is the plate equivalent to the weak form of Navier's hypothesis for beams (Bell, 2013). With the illustration in Fig. 2.9 and the assumption of small angles, this results in

$$tan(\theta) \approx \theta \qquad \Longrightarrow \qquad -u = z(-\theta) \qquad \Longleftrightarrow \qquad u = z\theta \qquad \text{(Eq. 2.6.33)}$$

The second assumption is that the center plane of the plate does not strain. These strains will be handled by the two-dimensional strains described in Chapter 2.6.2. The mathematical formulation can therefore be stated as



$$u_0 = u(x, y, 0) = 0 \qquad \text{(Eq. 2.6.34)}$$

$$v_0 = v(x, y, 0) = 0 \qquad \text{(Eq. 2.6.35)}$$

Eq. 2.6.33 in x and y direction gives

$$u = z\theta_y \qquad \text{(Eq. 2.6.36)}$$

$$v = -z\theta_x \qquad \text{(Eq. 2.6.37)}$$

Figure 2.9: Bending plate

These are functions of x and y

$$w = w(x, y) \qquad \theta_x = \theta_x(x, y) \qquad \theta_y = \theta_y(x, y) \qquad \text{(Eq. 2.6.38)}$$

Using these expressions, strain can be written as

$$\varepsilon_x = \frac{\partial u}{\partial x} = z\frac{\partial \theta_y}{\partial x} \tag{Eq. 2.6.39}$$

$$\varepsilon_y = \frac{\partial v}{\partial y} = -z\frac{\partial \theta_x}{\partial y} \tag{Eq. 2.6.40}$$

$$\varepsilon_z = \frac{\partial w}{\partial z} = 0 \tag{Eq. 2.6.41}$$

With the shear strain defined as

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} = z(\frac{\partial \theta_y}{\partial y} - \frac{\partial \theta_x}{\partial x}) \tag{Eq. 2.6.42}$$

Since the shear deformation is neglected due to thin plate theory, also known as Kirchhoff's hypothesis, the other shear strains can be set to zero. With the remaining terms relevant to bending, the plate strains can be written as

$$\boldsymbol{\varepsilon}_b = \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{bmatrix} = -z \begin{bmatrix} -\dfrac{\partial \theta_y}{\partial x} \\ \dfrac{\partial \theta_x}{\partial y} \\ \dfrac{\partial \theta_x}{\partial x} - \dfrac{\partial \theta_y}{\partial y} \end{bmatrix} = -z\mathbf{c} \tag{Eq. 2.6.43}$$

where $\quad \theta_x = \dfrac{\partial w}{\partial y} \quad$ and $\quad \theta_y = -\dfrac{\partial w}{\partial x}$

Eq. 2.6.43 can be reformulated as

$$\boldsymbol{\varepsilon}_b = -z \begin{bmatrix} \dfrac{\partial^2 w}{\partial x^2} \\ \dfrac{\partial^2 w}{\partial y^2} \\ \dfrac{\partial^2 w}{\partial x \partial y} \end{bmatrix} = -z\mathbf{c}_K \tag{Eq. 2.6.44}$$

The subscript K in the Kirchhoff curvature $\mathbf{c}_K$ indicates a Cartesian coordinate system. It is assumed that stresses in z direction is zero, even though this is not the actual case when strains are also zero in z direction. However, this discrepancy is insignificant enough to neglect (Bell, 2013).

The stress-strain relation can now be written as

$$\boldsymbol{\sigma}_b = \mathbf{C}_b \boldsymbol{\varepsilon}_b \tag{Eq. 2.6.45}$$

$\mathbf{C}_b$ is the same *elasticity matrix* as was found in Eq. 2.6.30. The stress resultants, which are the moments in Fig. 2.10, can now be calculated from Eq. 2.6.45 and Eq. 2.6.43

$$\mathbf{m} = \begin{bmatrix} M_x \\ M_y \\ M_{xy} \end{bmatrix} = \int_{-h/2}^{h/2} \boldsymbol{\sigma}_b z dz = -\mathbf{C}_b \int_{-h/2}^{h/2} z^2 dz \mathbf{c} = -\frac{h^3}{12} \mathbf{C}_b \mathbf{c} = -\mathbf{D}\mathbf{c} \tag{Eq. 2.6.46}$$

Here $\mathbf{D}$ is the *flexural rigidity* matrix for the plate.

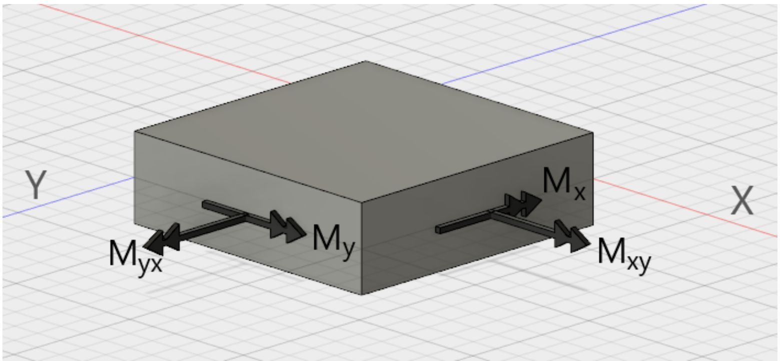
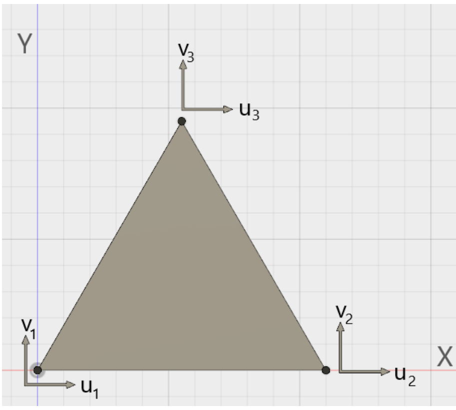


Figure 2.10: Membrane bending forces in plate element

2.6.4 CST - Constant Strain Triangle

The Constant Strain (and Stress) Triangle will represent the membrane forces in a shell element. The stress and strain vary linearly over the element as the displacement field is bi-linear and only deforms at the three edge nodes. Each node has two dofs, namely translation in x and y direction, which leaves the element with a total of 6 dofs. A method of indirect interpolation by shape functions will be used to establish the element membrane stiffness matrix.

The bi-linear displacement functions are set up for x and y direction as



$$u(x, y) = a_1 + a_2 x + a_3 y \qquad \text{(Eq. 2.6.47)}$$

$$v(x, y) = b_1 + b_2 x + b_3 y \qquad \text{(Eq. 2.6.48)}$$

Figure 2.11: CST bi-linear displacement field

Proceeding with Eq. 2.6.47, the equations for each node displacement can be written as

$$u_1 = a_1 + a_2 x_1 + a_3 y_1 \qquad \text{(Eq. 2.6.49)}$$

$$u_2 = a_1 + a_2 x_2 + a_3 y_2 \qquad \text{(Eq. 2.6.50)}$$

$$u_3 = a_1 + a_2 x_3 + a_3 y_3 \qquad \text{(Eq. 2.6.51)}$$

Which in matrix form is

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}}_{\mathbf{\Gamma}} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \mathbf{\Gamma a} \qquad \text{(Eq. 2.6.52)}$$

If Eq. 2.6.52 is solved for **a**, the expression becomes

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \frac{1}{\varrho} \begin{bmatrix} x_2y_3 - x_3y_2 & x_3y_1 - x_1y_3 & x_1y_2 - x_2y_1 \\ y_2 - y_3 & y_3 - y_1 & y_1 - y_2 \\ x_3 - x_2 & x_1 - x_3 & x_2 - x_1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \boldsymbol{\Gamma}^{-1}\mathbf{u}$$

(Eq. 2.6.53)

where $\varrho = x_1y_2 - x_2y_1 - x_1y_3 + x_3y_1 + x_2y_3 - x_3y_2$

Eq. 2.6.47 can now be expanded into

$$u(x,y) = a_1 + a_2x + a_3y = \begin{bmatrix} 1 & x & y \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & x & y \end{bmatrix} \Gamma^{-1} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad \text{(Eq. 2.6.54)}$$

The strain expression from Eq. 2.6.39 combined with Eq. 2.6.54 can thus be written as

$$\varepsilon_x = \frac{\partial u(x,y)}{\partial x} = \frac{\partial}{\partial x} \begin{bmatrix} 1 & x & y \end{bmatrix} \Gamma^{-1}\mathbf{u} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \Gamma^{-1} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \quad \text{(Eq. 2.6.55)}$$

The same procedure for v(x,y) in y direction gives

$$\varepsilon_y = \frac{\partial v(x,y)}{\partial y} = \frac{\partial}{\partial y} \begin{bmatrix} 1 & x & y \end{bmatrix} \Gamma^{-1}\mathbf{v} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \Gamma^{-1} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad \text{(Eq. 2.6.56)}$$

And the shear strain becomes

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \Gamma^{-1} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \Gamma^{-1} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \quad \text{(Eq. 2.6.57)}$$

There are three shape functions in a Constant Stress Triangle, one for each node, and all of them can be found as

$$\varepsilon_x = \frac{\partial}{\partial x}\underbrace{\begin{bmatrix}1 & x & y\end{bmatrix}\Gamma^{-1}}_{\mathbf{N}}\begin{bmatrix}u_1\\u_2\\u_3\end{bmatrix} = \frac{\partial}{\partial x}\mathbf{Nu} \qquad \text{(Eq. 2.6.58)}$$

Here **N** is the displacement field. The exact same would be found for v, the shape functions

$$\mathbf{N}^T = \begin{bmatrix}N_1\\N_2\\N_3\end{bmatrix} = \frac{1}{\varrho}\begin{bmatrix}x_2y_3 - x_3y_2 + x(y_2 - y_3) + y(x_3 - x_2)\\x_3y_1 - x_1y_3 + x(y_3 - y_1) + y(x_1 - x_3)\\x_1y_2 - x_2y_1 + x(y_1 - y_2) + y(x_2 - x_1)\end{bmatrix} \qquad \text{(Eq. 2.6.59)}$$

$$\text{where } \varrho = x_1y_2 - x_2y_1 - x_1y_3 + x_3y_1 + x_2y_3 - x_3y_2 \qquad \text{(Eq. 2.6.60)}$$

It is also interesting that the area of the triangle can be found as

$$\varrho = det(\mathbf{\Gamma}) = 2A \qquad \text{(Eq. 2.6.61)}$$

The total displacement vector will be rearranged according to the node numbering as

$$\mathbf{d} = \begin{bmatrix}u_1 & v_1 & u_2 & v_2 & u_3 & v_3\end{bmatrix}^T \qquad \text{(Eq. 2.6.62)}$$

The matrix $\mathbf{B}_m$, relating the strains and displacements, is defined as

$$\begin{bmatrix}\varepsilon_x\\\varepsilon_y\\\gamma_{xy}\end{bmatrix} = \underbrace{\begin{bmatrix}\frac{\partial}{\partial x} & 0\\0 & \frac{\partial}{\partial y}\\\frac{\partial}{\partial y} & \frac{\partial}{\partial x}\end{bmatrix}\begin{bmatrix}N_1 & 0 & N_2 & 0 & N_3 & 0\\0 & N_1 & 0 & N_2 & 0 & N_3\end{bmatrix}}_{\mathbf{B}_m}\begin{bmatrix}u_1\\v_1\\u_2\\v_2\\u_3\\v_3\end{bmatrix} \qquad \text{(Eq. 2.6.63)}$$

And thus

$$\mathbf{B}_m = \begin{bmatrix}\frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0\\0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & 0 & \frac{\partial N_3}{\partial y}\\\frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial y} & \frac{\partial N_3}{\partial x}\end{bmatrix} \qquad \text{(Eq. 2.6.64)}$$

Fully written out, this results in

$$\mathbf{B}_m = \frac{1}{2A}\begin{bmatrix} y_{23} & 0 & y_{31} & 0 & y_{12} & 0 \\ 0 & x_{32} & 0 & x_{13} & 0 & x_{21} \\ x_{32} & y_{23} & x_{13} & y_{31} & x_{21} & y_{12} \end{bmatrix} \qquad \text{(Eq. 2.6.65)}$$

$$x_{ij} = x_i - x_j \qquad\qquad y_{ij} = y_i - y_j$$

In this case, the entries in $\mathbf{B}_m$ are constants. This is not necessarily the case for higher order displacement polynomial elements.

Through use of the principal of virtual displacement an expression for the element stiffness matrix can be established (Bell, 2013). It can be expressed as

$$\mathbf{k}^e = \int_{V_e} \mathbf{B}^T \mathbf{C} \mathbf{B} dV \qquad \text{(Eq. 2.6.66)}$$

C in this case is the matrix found in Eq. 2.6.30, and **B** is the newly derived matrix from Eq. 2.6.64. If a constant thickness of t is assumed for the plate, Eq. 2.6.66 becomes

$$\mathbf{k}^e_m = \int_{A_e} \mathbf{B}^T_m \mathbf{C} \mathbf{B}_m t dA \qquad \text{(Eq. 2.6.67)}$$

If both $\mathbf{B}_m$ and **C** are independent of the area, Eq. 2.6.67 simplifies to

$$\mathbf{k}^e_m = At\mathbf{B}^T_m \mathbf{C} \mathbf{B}_m \qquad \text{(Eq. 2.6.68)}$$

In matrix form, this becomes

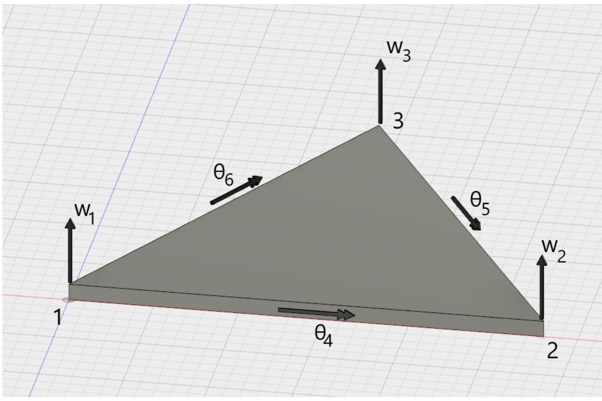
$$\mathbf{k}^e_m = \frac{Et}{4A(1-\nu^2)} \begin{bmatrix} y_{23} & 0 & x_{32} \\ 0 & x_{32} & y_{23} \\ y_{31} & 0 & x_{13} \\ 0 & x_{13} & y_{31} \\ y_{12} & 0 & x_{21} \\ 0 & x_{21} & y_{12} \end{bmatrix} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{bmatrix} y_{23} & 0 & y_{31} & 0 & y_{12} & 0 \\ 0 & x_{32} & 0 & x_{13} & 0 & x_{21} \\ x_{32} & y_{23} & x_{13} & y_{31} & x_{21} & y_{12} \end{bmatrix}$$

(Eq. 2.6.69)

where $x_{ij} = x_i - x_j \qquad y_{ij} = y_i - y_j$

2.6.5 The Morley Triangle

The Morley triangle is the simplest triangular plate bending element attainable according to Bell (2013), and only has three nodes and six dofs. Three of the dofs are the translations out of plane, while the remaining three dofs gives the rotation around each side of the triangle, as illustrated in Fig. 2.12. The dofs are given as



$$\mathbf{v} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \theta_4 \\ \theta_5 \\ \theta_6 \end{bmatrix} \qquad \text{(Eq. 2.6.70)}$$

Figure 2.12: Dofs for the Morley triangle

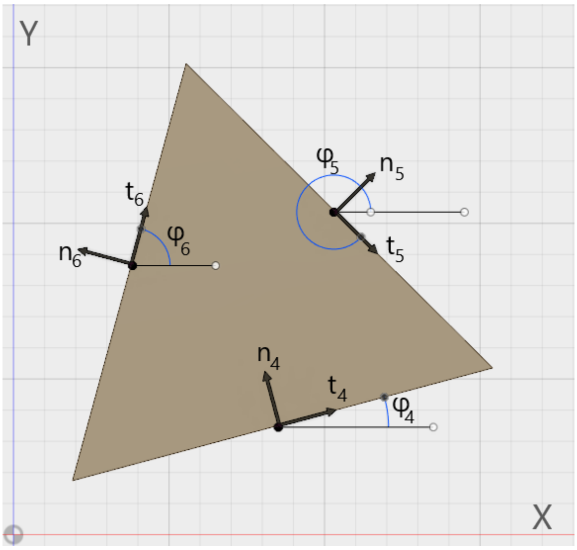
The Morley triangle has its base in a quadratic polynomial. It satisfies the completeness criteria for shape functions, but does not satisfy continuity (Bell, 2013). Despite this, the element behaves rather well according to Bell (2013), which in combination with the low amount of dofs is the reason it has been selected for implementation. The area coordinates described in Ch. 2.6.1 through indirect interpolation will be utilized to ease the process for establishing the element stiffness matrix for bending. Despite the Morley triangle being a basic element, this is not a minor task.

From a complete quadratic polynomial, the equivalent homogeneous polynomial is assumed in area coordinates as

$$w = \begin{bmatrix} \zeta_1^2 & \zeta_2^2 & \zeta_3^2 & \zeta_1\zeta_2 & \zeta_2\zeta_3 & \zeta_3\zeta_1 \end{bmatrix} = \mathbf{N_g g} \qquad \text{(Eq. 2.6.71)}$$

Here, **g** is the generalized displacement parameters. The relation between w and **v** is now needed, which can be done by expressing **v** through **g**. First, however, the rotations must be defined in area coordinates. If the independent variables are chosen like in Eq. 2.6.16, the independent variables are ζ_1 and ζ_2.

The rotations now need to be expressed through derivatives with respect to ζ_1 and ζ_2, but to do this some unambiguous expressions for the normal slope θ_m must be established. Note that the directions for the rotations in Fig. 2.12 and the t axes in Fig. 2.13 is oriented towards the positive local x axis for the element. Through inspection of Fig. 2.13, it can be stated that



$$0 \leq \phi_m < \frac{\pi}{2} \qquad \text{(Eq. 2.6.72)}$$

or

$$\frac{3\pi}{2} \leq \phi_m < 2\pi \qquad \text{(Eq. 2.6.73)}$$

where $\quad m = 4, 5, 6$

Figure 2.13: Normal slope definition

Notation for cosine and sine is then denoted as

$$c_m \equiv cos(\phi_m) \qquad s_m \equiv sin(\phi_m)$$

The relationship between the x-y coordinates and n-t coordinates can through further inspection of Fig. 2.13 be defined as follows

$$x = ct - sn \qquad y = st + cn \qquad \text{(Eq. 2.6.74)}$$

$$t = cx + sy \qquad n = -sx + cy \qquad \text{(Eq. 2.6.75)}$$

The derivatives can be expressed as

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial}{\partial y}\frac{\partial y}{\partial t} = c\frac{\partial}{\partial x} + s\frac{\partial}{\partial y} \quad \text{(Eq. 2.6.76)}$$

$$\frac{\partial}{\partial n} = \frac{\partial}{\partial x}\frac{\partial x}{\partial n} + \frac{\partial}{\partial y}\frac{\partial y}{\partial n} = -s\frac{\partial}{\partial x} + c\frac{\partial}{\partial y} \quad \text{(Eq. 2.6.77)}$$

The rotations from Eq. 2.6.70 can now be expressed as

$$\theta_m = \frac{\partial w_m}{\partial n} = -s_m\frac{\partial w_m}{\partial x} + c_m\frac{\partial w_m}{\partial y} \quad \text{(Eq. 2.6.78)}$$

The element stiffness relation in Eq. 2.6.66 was defined as

$$\mathbf{k}^e = \int_{V_e} \mathbf{B}^T\mathbf{C}\mathbf{B}dV \quad \text{(Eq. 2.6.79)}$$

The **B** matrix is missing for bending, but can be established from the basic assumption that

$$\varepsilon = \Delta\mathbf{u} = \Delta\mathbf{N}\mathbf{v} = \mathbf{B}\mathbf{v} \quad \text{(Eq. 2.6.80)}$$

Here **u** is the displacement component vector relating to ”real” strain. Remember that **v** is locally defined.

The bending strain can with Eq. 2.6.80 combined with Eq. 2.6.44 and Eq. 2.6.71 be written as

$$\begin{aligned}\varepsilon_b &= -z\mathbf{c}_K \\ &= -z\Delta_K w \\ &= -z\Delta_K\mathbf{N}_g\mathbf{g} \\ &= -z\Delta_K\mathbf{N}_g\mathbf{A}^{-1}\mathbf{v} \\ &= -z\mathbf{B}_K\mathbf{v}\end{aligned} \quad \text{(Eq. 2.6.81)}$$

Solving for **B** gives

$$\mathbf{B} = -z\mathbf{B}_K = -z\Delta_K\mathbf{N}_g\mathbf{A}^{-1} \quad \text{(Eq. 2.6.82)}$$

And the sought relation between **v** and **g** is given by the **A** matrix as

$$\mathbf{g} = \mathbf{A}^{-1}\mathbf{v} \quad \text{and} \quad \mathbf{v} = \mathbf{A}\mathbf{g} \quad \text{(Eq. 2.6.83)}$$

If the term for **B** from Eq. 2.6.82 is substituted into Eq. 2.6.79, the element bending stiffness matrix can be written as

$$\mathbf{k}_b^e = \int_{-h/2}^{h/2} \int_{A_e} (-z\mathbf{B}_K^T)\mathbf{C}_b(-z\mathbf{B}_K) dz dA = \frac{1}{12} \int_{A_e} h^3 \mathbf{B}_K^T \mathbf{C}_b \mathbf{B}_K dA \quad \text{(Eq. 2.6.84)}$$

or, with constant plate thickness, simply

$$\mathbf{k}_b^e = \int_{A_e} \mathbf{B}_K^T \mathbf{D} \mathbf{B}_K dA \qquad \text{where} \qquad \mathbf{D} = \frac{h^3}{12}\mathbf{C}_b \quad \text{(Eq. 2.6.85)}$$

With an expression for the stiffness matrix for bending established, the next step will be to determine the **A** matrix. As the shape functions are in terms of area coordinates, an expression for the normal slope θ_m derived with respect to area coordinates is needed. Through Eq. 2.6.17 the relation from Eq. 2.6.78 can be written as

$$\theta_m = \frac{\partial w_m}{\partial n} \quad \text{(Eq. 2.6.86)}$$

$$= \begin{bmatrix} -s_m & c_m \end{bmatrix} \begin{bmatrix} \dfrac{\partial w_m}{\partial x} \\ \dfrac{\partial w_m}{\partial y} \end{bmatrix} \quad \text{(Eq. 2.6.87)}$$

$$= \begin{bmatrix} -s_m & c_m \end{bmatrix} \frac{1}{2A} \begin{bmatrix} y_{23} & y_{31} \\ x_{32} & x_{13} \end{bmatrix} \begin{bmatrix} \dfrac{\partial w_m}{\partial \zeta_1} \\ \dfrac{\partial w_m}{\partial \zeta_2} \end{bmatrix} \quad \text{(Eq. 2.6.88)}$$

$$= \frac{c_m x_{32} - s_m y_{23}}{2A} \frac{\partial w_m}{\partial \zeta_1} + \frac{c_m x_{13} - s_m y_{31}}{2A} \frac{\partial w_m}{\partial \zeta_2} \quad \text{(Eq. 2.6.89)}$$

For simplicity, the following notation is introduced

$$\gamma_m = \frac{c_m x_{32} - s_m y_{23}}{2A} \quad \text{(Eq. 2.6.90)}$$

$$\mu_m = \frac{c_m x_{13} - s_m y_{31}}{2A} \quad \text{(Eq. 2.6.91)}$$

$$\alpha_m = \gamma_m + \mu_m \quad \text{(Eq. 2.6.92)}$$

Going through the nodes of the element and knowing that $\zeta_3 = 1 - \zeta_1 - \zeta_2$ and $\zeta_2 = 0$ at node 1, and so on, it can from Eq. 2.6.71 be shown that

$$v_1 = w_1 = g_1$$
$$v_2 = w_2 = g_2$$
$$v_3 = w_3 = g_3$$

At "mid-edge node" 4 it becomes $\zeta_3 = 0$ and $\zeta_1 = \zeta_2 = 1/2$, and similarly for 5 and 6. This calculation is a tedious task to do by hand, therefore the Matlab script shown in Lst. 2.1 was used to perform the derivation of these expressions.

```
syms g4 m4 g5 m5 g6 m6 z1 z2;
z3 = 1 - z1 - z2;
N = [z1^2 z2^2 z3^2 z1*z2 z2*z3 z3*z1];

dw4 = g4*diff(N,z1) + m4*diff(N,z2);
dw5 = g5*diff(N,z1) + m5*diff(N,z2);
dw6 = g6*diff(N,z1) + m6*diff(N,z2);

v4 = subs(dw4,[z1,z2], [1/2,1/2]);
v5 = subs(dw5,[z1,z2], [0,1/2]);
v6 = subs(dw6,[z1,z2], [1/2,0]);
```

Listing 2.1: Deriving equations for v_4, v_5 and v_6

Running this script gives the equations

$$v_4 = \gamma_4 g_1 + \mu_4 g_2 + \frac{1}{2}\alpha_4 g_4 - \frac{1}{2}\alpha_4 g_5 - \frac{1}{2}\alpha_4 g_6$$

$$v_5 = \mu_5 g_2 - \alpha_5 g_3 + \frac{1}{2}\gamma_5 g_4 - \frac{1}{2}\gamma_5 g_5 + \frac{1}{2}\gamma_5 g_6$$

$$v_6 = \gamma_6 g_1 - \alpha_6 g_3 + \frac{1}{2}\mu_6 g_4 + \frac{1}{2}\mu_6 g_5 - \frac{1}{2}\mu_6 g_6$$

From Eq. 2.6.83, the $\mathbf{A}$ matrix can now be established as

$$\mathbf{v} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \theta_4 \\ \theta_5 \\ \theta_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \gamma_4 & \mu_4 & 0 & \frac{\alpha_4}{2} & \frac{-\alpha_4}{2} & \frac{\alpha_4}{2} \\ 0 & \mu_5 & -\alpha_5 & \frac{\gamma_5}{2} & \frac{-\gamma_5}{2} & \frac{\gamma_5}{2} \\ \gamma_6 & 0 & -\alpha_6 & \frac{\mu_6}{2} & \frac{\mu_6}{2} & \frac{-\mu_6}{2} \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ g_4 \\ g_5 \\ g_6 \end{bmatrix} = \mathbf{Ag} \qquad \text{(Eq. 2.6.93)}$$

In matrix notation, this can be written as

$$\mathbf{v} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \mathbf{g} \qquad \text{(Eq. 2.6.94)}$$

When inverted, $\mathbf{A}$ becomes

$$\mathbf{A}^{-1} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{A}_{22}^{-1}\mathbf{A}_{21} & \mathbf{A}_{22}^{-1} \end{bmatrix} \qquad \text{(Eq. 2.6.95)}$$

Having established the $\mathbf{A}$ matrix, there is a problem of coordinate system from Eq. 2.6.82 where $\mathbf{N}_g$ is given in area coordinates and Δ_K is in Cartesian coordinates. By applying Eq. 2.6.17 twice, the following transition can be found

$$\begin{bmatrix} \dfrac{\partial^2}{\partial x^2} \\ \dfrac{\partial^2}{\partial y^2} \\ 2\dfrac{\partial^2}{\partial x \partial y} \end{bmatrix} = \underbrace{\frac{1}{4A^2} \begin{bmatrix} y_{23}^2 & y_{31}^2 & 2y_{23}y_{31} \\ x_{32}^2 & x_{13}^2 & 2x_{13}x_{32} \\ 2x_{32}y_{23} & 2x_{13}y_{31} & 2(x_{13}y_{23} + x_{32}y_{31}) \end{bmatrix}}_{\mathbf{H}} \begin{bmatrix} \dfrac{\partial^2}{\partial \zeta_1^2} \\ \dfrac{\partial^2}{\partial \zeta_2^2} \\ \dfrac{\partial^2}{\partial \zeta_1 \partial \zeta_2} \end{bmatrix} \qquad \text{(Eq. 2.6.96)}$$

Which in short can be written as

$$\Delta_K = \mathbf{H}\Delta_\zeta \qquad \text{(Eq. 2.6.97)}$$

The expression for $\mathbf{B}_K$ from Eq. 2.6.82 can now be written as

$$\mathbf{B}_K = \Delta_K \mathbf{N}_g \mathbf{A}^{-1} = \mathbf{H} \underbrace{\Delta_\zeta \mathbf{N}_g}_{\mathbf{B}_g} \mathbf{A}^{-1} = \mathbf{H}\mathbf{B}_g\mathbf{A}^{-1} \qquad \text{(Eq. 2.6.98)}$$

$\mathbf{B}_g$ for this element is a constant matrix defined as

$$\mathbf{B}_g = \Delta_\zeta \begin{bmatrix} \zeta_1^2 & \zeta_2^2 & \zeta_3^2 & \zeta_1\zeta_2 & \zeta_2\zeta_3 & \zeta_3\zeta_1 \end{bmatrix}$$

$$= \begin{bmatrix} \dfrac{\partial^2}{\partial \zeta_1^2} \\ \dfrac{\partial^2}{\partial \zeta_2^2} \\ \dfrac{\partial^2}{\partial \zeta_1 \partial \zeta_2} \end{bmatrix} \begin{bmatrix} \zeta_1^2 & \zeta_2^2 & (1-\zeta_1-\zeta_2)^2 & \zeta_1\zeta_2 & \zeta_2(1-\zeta_1-\zeta_2) & (1-\zeta_1-\zeta_2)\zeta_1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 0 & 2 & 0 & 0 & -2 \\ 0 & 2 & 2 & 0 & -2 & 0 \\ 0 & 0 & 2 & 1 & -1 & -1 \end{bmatrix} \tag{Eq. 2.6.99}$$

All the terms that defines $\mathbf{B}_K$ are now known. The expression for the element bending stiffness matrix from Eq. 2.6.85 can then be evaluated as

$$\mathbf{k}_b^e = \int_{A_e} \mathbf{B}_K^T \mathbf{D} \mathbf{B}_K dA = \int_{A_e} \mathbf{A}^{-T} \mathbf{B}_g^T \mathbf{H}^T \mathbf{D} \mathbf{H} \mathbf{B}_g \mathbf{A}^{-1} dA \tag{Eq. 2.6.100}$$

For a defined triangular element all of these matrices are constants, which means the expression becomes

$$\mathbf{k}_b^e = \mathbf{A}^{-T} \mathbf{B}_g^T \mathbf{H}^T \mathbf{D} \mathbf{H} \mathbf{B}_g \mathbf{A}^{-1} A_e \tag{Eq. 2.6.101}$$

$$\text{where} \qquad A_e = \text{Area of triangle} \tag{Eq. 2.6.102}$$

The bending forces from Eq. 2.6.46 can in combination with Eq. 2.6.81 and Eq. 2.6.98 now be written as

$$\mathbf{m} = \begin{bmatrix} M_x \\ M_y \\ M_{xy} \end{bmatrix} = -\mathbf{D}\mathbf{c} = -\mathbf{D}\mathbf{B}_K \mathbf{v} = -\mathbf{D}\mathbf{H}\mathbf{B}_g \mathbf{A}^{-1} \mathbf{v} \tag{Eq. 2.6.103}$$

2.6.6 Triangular Shell Element Assembly

To acquire the element stiffness matrix for a shell element, a membrane element and a bending element can be assembled as

$$\mathbf{k^e}_{shell} = \begin{bmatrix} \mathbf{k}^e_m & \mathbf{0} \\ \mathbf{0} & \mathbf{k}^e_b \end{bmatrix} \quad \text{and} \quad \mathbf{v}_{shell} = \begin{bmatrix} \mathbf{v}_m \\ \mathbf{v}_b \end{bmatrix} \qquad \text{(Eq. 2.6.104)}$$

For an element consisting of a CST element as described in Ch. 2.6.4 for the membrane part and a Morley triangle element from Ch. 2.6.5 for the bending part, and the assembly becomes

$$\mathbf{k^e}_{shell} = \begin{bmatrix} \mathbf{k}^e_{m,CST} & \mathbf{0} \\ \mathbf{0} & \mathbf{k}^e_{b,Morley} \end{bmatrix} \qquad \text{(Eq. 2.6.105)}$$

$$\mathbf{v}_{shell} = \begin{bmatrix} u_1 & v_1 & u_2 & v_2 & u_3 & v_3 & w_1 & w_2 & w_3 & \theta_4 & \theta_5 & \theta_6 \end{bmatrix}^T \qquad \text{(Eq. 2.6.106)}$$

For the CST-Morley shell element it should be noted that it only has 12 dofs per element, as shown in Fig. 2.14. In other words, not more than a normal 3D Beam element. The dofs can be rearranged in any order, as long as this is taken into account when transforming from local to global and vice versa.

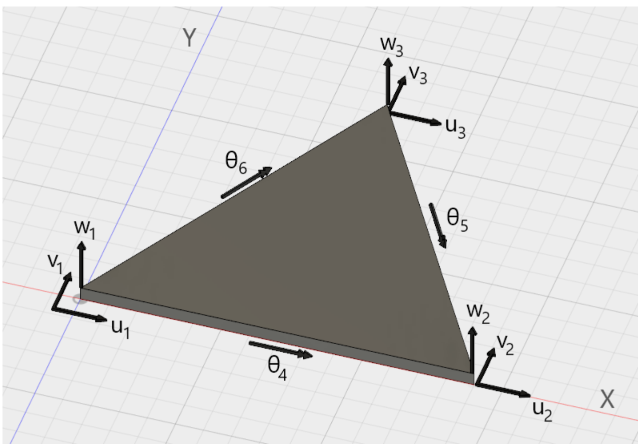


Figure 2.14: The degrees of freedom for the CST-Morley element

2.7 Transformation Matrix

Each element has a global (X, Y, Z) and a local (x, y, z) coordinate system. Stiffness (E, G, A, Ix, Iy, Iz, L) is evaluated in the local coordinate system, and are independent of the beam's location in global space. In order to relate an element's stiffness matrix to the global stiffness matrix, we must use a transformation matrix.

First we can define the transformation matrix $\mathbf{T}$ in such a way that

$$\delta = \mathbf{T\Delta} \qquad\qquad \mathbf{p} = \mathbf{TP} \tag{Eq. 2.7.1}$$

Here δ is a list of generalized unit displacement (in local coordinate system), $\mathbf{\Delta}$ is a list of generalized unit displacement (in global coordinate system), **p** is a list of local forces and **P** is a vector of global forces.

Clarification of notation: capital letters signifies stiffness matrix in global coordinates, while superscripted G signifies global stiffness matrix (unlike e for element). Matrices and vectors are written in **bold**, and node numbers are denoted by i.

By inserting Eq. 2.7.1 into Eq. 2.5.47 from Chapter 2.3 we obtain

$$\mathbf{TP} = \mathbf{k^e T\Delta} \tag{Eq. 2.7.2}$$

Premultiplying this with $\mathbf{T^{-1}}$ gives

$$\mathbf{P} = \mathbf{T^{-1} k^e T\Delta} \tag{Eq. 2.7.3}$$

Since the matrix $\mathbf{T}$ is orthogonal, the inverse and transposed will be identical, which means that

$$\mathbf{P} = \mathbf{T^T k^e T\Delta} \tag{Eq. 2.7.4}$$

$\mathbf{K^e}$ is the element in global coordinates, defined as

$$\mathbf{P} = \mathbf{K^e \Delta} \tag{Eq. 2.7.5}$$

This means the relationship between local and global coordinates can be defined as

$$\mathbf{K^e} = \mathbf{T^T k^e T} \tag{Eq. 2.7.6}$$

2D Transformation Matrix

The transformation matrix itself is constructed by rotation of the local axes. For a 3-dof system like in Figure 2.15, this means projecting the local dofs v_i (of node i) to the global dof V_i by sine and cosine.

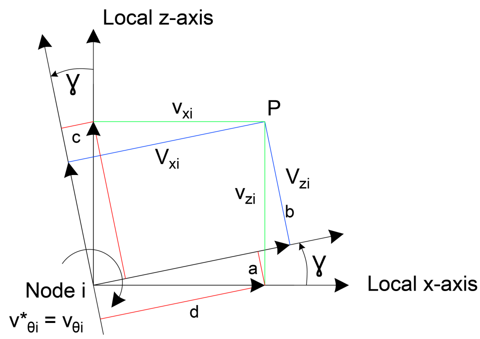


Figure 2.15: Transformation of axes in two dimensions

As can be seen on Fig. 2.15, the global translations V_{xi} and V_{zi} are defined as

$$a = v_{xi} \sin\gamma \qquad b = v_{zi} \cos\gamma \tag{Eq. 2.7.7}$$

$$c = v_{zi} \sin\gamma \qquad d = v_{xi} \cos\gamma \tag{Eq. 2.7.8}$$

$$V_{xi} = c + d \implies V_{xi} = v_{xi} \cos\gamma + v_{zi} \sin\gamma \tag{Eq. 2.7.9}$$

$$V_{zi} = b - a \implies V_{zi} = -v_{xi} \sin\gamma + v_{zi} \cos\gamma \tag{Eq. 2.7.10}$$

$$V_{\theta i} = v_{\theta i} \tag{Eq. 2.7.11}$$

Simplified notation gives

$$\cos\gamma = c \qquad \sin\gamma = s \tag{Eq. 2.7.12}$$

In matrix form

$$\mathbf{V_i} = \begin{bmatrix} V_{xi} \\ V_{zi} \\ V_{\theta i} \end{bmatrix} = \begin{bmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{xi} \\ v_{zi} \\ v_{\theta i} \end{bmatrix} = \mathbf{t v_i} \tag{Eq. 2.7.13}$$

For 2-noded elements (like in Figure 2.16) the transformation matrix becomes

$$\mathbf{V} = \begin{bmatrix} \mathbf{V_1} \\ \mathbf{V_2} \end{bmatrix} = \begin{bmatrix} \mathbf{t} & \mathbf{0} \\ \mathbf{0} & \mathbf{t} \end{bmatrix} \begin{bmatrix} \mathbf{v_1} \\ \mathbf{v_2} \end{bmatrix} = \mathbf{Tv} \qquad \text{(Eq. 2.7.14)}$$

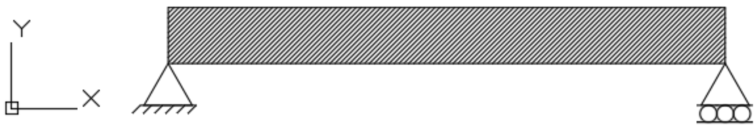


Figure 2.16: A simply supported beam

For a simple beam with 3 dofs per node, the $\mathbf{k}^e$ can be constructed by combining Eq. 2.5.57 and Eq. 2.5.66, and the transformation matrix $\mathbf{T}$ as shown in Eq. 2.7.14.

$$\mathbf{k^e} = \begin{bmatrix} \mu & 0 & 0 & -\mu & 0 & 0 \\ 0 & 12 & -6L & 0 & -12 & -6L \\ 0 & -6L & 4L^2 & 0 & 6L & 2L^2 \\ -\mu & 0 & 0 & \mu & 0 & 0 \\ 0 & -12 & 6L & 0 & 12 & 6L \\ 0 & -6L & 2L^2 & 0 & 6L & 4L^2 \end{bmatrix} \frac{EI}{L^3} \qquad \text{where } \mu = \frac{AL^2}{I}$$

(Eq. 2.7.15)

$$\mathbf{V} = \begin{bmatrix} V_{xi} \\ V_{zi} \\ V_{\theta i} \\ V_{xi+1} \\ V_{zi+1} \\ V_{\theta i+1} \end{bmatrix} = \begin{bmatrix} c & s & 0 & 0 & 0 & 0 \\ -s & c & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & c & s & 0 \\ 0 & 0 & 0 & -s & c & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{xi} \\ v_{zi} \\ v_{\theta i} \\ v_{xi+1} \\ v_{zi+1} \\ v_{\theta i+1} \end{bmatrix} = \mathbf{Tv} \qquad \text{(Eq. 2.7.16)}$$

As known from in Eq. 2.7.6, the expression for the element stiffness matrix in global coordinates is

$$\mathbf{K^e} = \mathbf{T^T k^e T}$$

This leads to

$$\mathbf{K^e} = \frac{EI}{L^3} \cdot \begin{bmatrix} \mu c^2 + 12s^2 & \mu cs - 12cs & 6Ls & -\mu c^2 - 12s^2 & -\mu cs + 12cs & 6Ls \\ \mu cs - 12cs & \mu s^2 + 12c^2 & -6Lc & -\mu cs + 12cs & -\mu s^2 - 12c^2 & -6Lc \\ 6Ls & -6Lc & 4L^2 & -6Ls & 6Lc & 2L^2 \\ -\mu c^2 - 12s^2 & -\mu cs + 12cs & -6Ls & \mu c^2 + 12s^2 & \mu cs - 12cs & -6Ls \\ -\mu cs + 12cs & -\mu s^2 - 12c^2 & 6Lc & \mu cs - 12cs & \mu s^2 + 12c^2 & 6Lc \\ 6Ls & -6Lc & 2L^2 & -6Ls & 6Lc & 4L^2 \end{bmatrix}$$

(Eq. 2.7.17)

Which when fully written out gives the element stiffness matrix in the global coordinate system. μ is defined in Eq. 2.7.16.

3D Transformation Matrix

For simple 3D coordinate transformation the direction cosines can be utilized to transform from global coordinates x_g, y_g, z_g to local x_l, y_l, z_l as

$$\mathbf{v}_l = \begin{bmatrix} u_{xl} \\ u_{yl} \\ u_{zl} \end{bmatrix} = \begin{bmatrix} c(x_l, x_g) & c(x_l, y_g) & c(x_l, z_g) \\ c(y_l, x_g) & c(y_l, y_g) & c(y_l, z_g) \\ c(z_l, x_g) & c(z_l, y_g) & c(z_l, z_g) \end{bmatrix} \begin{bmatrix} u_{xg} \\ u_{yg} \\ u_{zg} \end{bmatrix} = \mathbf{T}\mathbf{v}_l \qquad \text{(Eq. 2.7.18)}$$

where $c(x_l, x_g)$ is the cosine of the angle between the local x axis x_l and the global x axis x_g. The direction cosines is therefore dependent upon having the three local axes defined. The directional cosines can be observed on Fig. 2.18a. Without the defined local axes this becomes more of a challenge, and for beams a more handy transformation matrix can be derived as follows.

For beams, allowing for rotation about its local x-axis requires adding the angle α as shown in Figure 2.17, while allowing for simple 3D rotation requires adding an angle β and γ as shown in Figure 2.18b

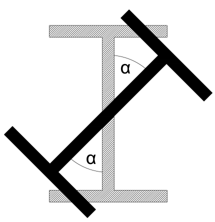
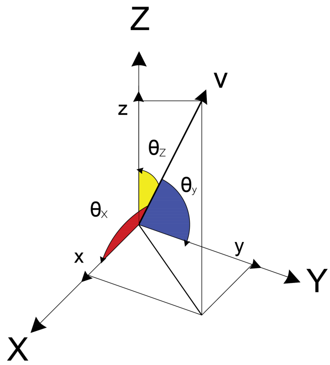
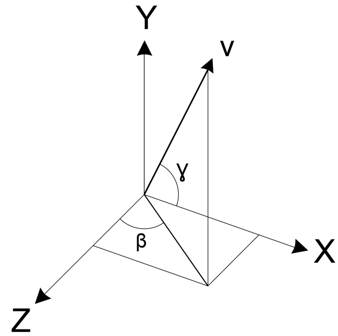


Figure 2.17: Rotation α about local x-axis



(a) Direction angles



(b) Rotation β about Y-axis and γ about Z-axis

Figure 2.18: Angles needed for transformation in arbitrary 3D coordinates

The general case must take all three angles into account.

$$\mathbf{t} = \begin{bmatrix} \mathbf{R}_\gamma & \mathbf{R}_\beta & \mathbf{R}_\alpha \end{bmatrix} \qquad \text{(Eq. 2.7.19)}$$

Similarly to for Eq. 2.7.9-2.7.11, the angles between the different axes can be described by sine and cosine. Following the same procedure as the 2D case, the rotational matrices becomes

$$\mathbf{R}_\gamma = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{(Eq. 2.7.20)}$$

$$\mathbf{R}_\beta = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \qquad \text{(Eq. 2.7.21)}$$

$$\mathbf{R}_\alpha = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \qquad \text{(Eq. 2.7.22)}$$

Next, it would be beneficial to describe the angles in terms of directional cosines instead of angles, since that makes them easy to calculate for a line element. Directional cosines are defined as the cosines of angles between two vectors and are the component's length contribution per unit vector in that direction.

$$C_X = \cos\theta_x = \frac{x_j - x_i}{L} \qquad C_Y = \cos\theta_y = \frac{y_j - y_i}{L} \qquad C_Z = \cos\theta_z = \frac{z_j - z_i}{L} \tag{Eq. 2.7.23}$$

$$L = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2} \qquad C_{XZ} = \sqrt{C_X^2 + C_Z^2} \tag{Eq. 2.7.24}$$

Note that

$$\sin\gamma = C_Y \qquad \cos\gamma = C_{XZ} \tag{Eq. 2.7.25}$$

$$\sin\beta = \frac{C_Z}{C_{XZ}} \qquad \cos\beta = \frac{C_X}{C_{XZ}} \tag{Eq. 2.7.26}$$

Multiplication of these matrices yields Matrix 2.7.27.

$$\mathbf{t} = \begin{bmatrix} C_X & C_Y & C_Z \\ \frac{-C_X C_Y \cos\alpha - C_Z \sin\alpha}{C_{XZ}} & C_{XZ}\cos\alpha & \frac{-C_Y C_Z \cos\alpha + C_X \sin\alpha}{C_{XZ}} \\ \frac{C_X C_Y \sin\alpha - C_Z \cos\alpha}{C_{XZ}} & -C_{XZ}\sin\alpha & \frac{C_Y C_Z \sin\alpha + C_X \cos\alpha}{C_{XZ}} \end{bmatrix} \tag{Eq. 2.7.27}$$

Beware that some entries are divided by C_{XZ} which is zero if the nodal points only change along the Y-axis (i.e. $x_j - x_i$ = 0 and $z_j - z_i = 0$). For this case, C_X, C_Z and C_{XZ} are all zero, so **t** = $\mathbf{R}_\gamma \mathbf{R}_\alpha$. Since $\mathbf{R}_\gamma$ is simplified to

$$\begin{bmatrix} 0 & C_Y & 0 \\ -C_Y & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{Eq. 2.7.28}$$

The matrix **t** simplifies to

$$\mathbf{t} = \begin{bmatrix} 0 & C_Y & 0 \\ -C_Y\cos\alpha & 0 & \sin\alpha \\ C_Y \sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{Eq. 2.7.29}$$

2.8 Global Stiffness Matrix

After the element stiffness matrix has been converted to global coordinates, they must be assembled into the global stiffness matrix. The global stiffness matrix consists of stiffnesses from all global dofs (gdofs) and is a gdof by gdof matrix.

The procedure for assembling the global stiffness matrix is as follows:

1. Construct element stiffness matrix $\mathbf{k^e}$ for each element.
2. Transform element matrices to global coordinates $\mathbf{K^e}$.
3. Enter stiffnesses from $\mathbf{K^e}$ into correct entries in global stiffness matrix $\mathbf{K^G}$. When nodes are shared among elements, stiffnesses are summed.

A pseudocode for the procedure is shown in Lst. 2.2.

```
foreach element in elements
    ke = Get local element stiffness matrix
    T = Get element transformation matrix
    Ke = T^T*ke*T

    index1 = Get index of node 1 in Point List
    index2 = Get index of node 2 in Point List

    KG(index1, index1) = KG(index1, index1) + Ke(1,1)
    KG(index1, index2) = KG(index1, index2) + Ke(1,2)
    KG(index2, index1) = KG(index2, index1) + Ke(2,1)
    KG(index2, index2) = KG(index2, index2) + Ke(2,2)
```

Listing 2.2: Pseudocode for assembly of KG

The global stiffness matrix must take into account the stiffnesses from all elements, which means that any elements that shares nodes must sum their stiffnesses. As an example, Step 3 has been performed for two element matrices identical to the one from Eq. 2.7.17 and is shown in Eq. 2.8.1. Observe that entries in rows and columns 4-6 contain summed stiffness.

$$\mathbf{K^G} = \begin{bmatrix} K^1_{1,1} & K^1_{1,2} & K^1_{1,3} & K^1_{1,4} & K^1_{1,5} & K^1_{1,6} & 0 & 0 & 0 \\ K^1_{2,1} & K^1_{2,2} & K^1_{2,3} & K^1_{2,4} & K^1_{2,5} & K^1_{2,6} & 0 & 0 & 0 \\ K^1_{3,1} & K^1_{3,2} & K^1_{3,3} & K^1_{3,4} & K^1_{3,5} & K^1_{3,6} & 0 & 0 & 0 \\ K^1_{4,1} & K^1_{4,2} & K^1_{4,3} & K^1_{4,4}+K^2_{1,1} & K^1_{4,5}+K^2_{1,2} & K^1_{4,6}+K^2_{1,3} & K^2_{1,4} & K^2_{1,5} & K^2_{1,6} \\ K^1_{5,1} & K^1_{5,2} & K^1_{5,3} & K^1_{5,4}+K^2_{2,1} & K^1_{5,5}+K^2_{2,2} & K^1_{5,6}+K^2_{2,3} & K^2_{2,4} & K^2_{2,5} & K^2_{2,6} \\ K^1_{6,1} & K^1_{6,2} & K^1_{6,3} & K^1_{6,4}+K^2_{3,1} & K^1_{6,5}+K^2_{3,2} & K^1_{6,6}+K^2_{3,3} & K^2_{3,4} & K^2_{3,5} & K^2_{3,6} \\ 0 & 0 & 0 & K^2_{4,1} & K^2_{4,2} & K^2_{4,3} & K^2_{4,4} & K^2_{4,5} & K^2_{4,6} \\ 0 & 0 & 0 & K^2_{5,1} & K^2_{5,2} & K^2_{5,3} & K^2_{5,4} & K^2_{5,5} & K^2_{5,6} \\ 0 & 0 & 0 & K^2_{6,1} & K^2_{6,2} & K^2_{6,3} & K^2_{6,4} & K^2_{6,5} & K^2_{6,6} \end{bmatrix}$$

(Eq. 2.8.1)

2.9 Cholesky Banachiewicz

The Cholesky decomposition method can be used to numerically solve matrices of the form **Ax = b**. The method works by first decomposing **A** into the lower triangular matrix **L** and its conjugate transposed. **L** is then used to calculate **y** by forward substitution. And finally, **x** can be found by performing back substitution on **y**. Note that the conjugate transposed matrix will be identical to the transposed matrix when only dealing with real numbers.

$$\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}} \qquad \implies \mathbf{L}\mathbf{y} = \mathbf{b} \qquad \implies \mathbf{L}^{\mathbf{T}}\mathbf{x} = \mathbf{y} \qquad \text{(Eq. 2.9.1)}$$

To show how $\mathbf{L}$ and $\mathbf{L}^{\mathbf{T}}$ can be found, consider a square, symmetric 2x2 matrix, **A**. Since $\mathbf{L}^{\mathbf{T}}$ is the transpose of $\mathbf{L}$, they are always symmetric to each other.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2 & 8 \end{bmatrix} = \mathbf{L}\mathbf{L}^{\mathbf{T}} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} \\ 0 & L_{22} \end{bmatrix} = \begin{bmatrix} L_{11}^2 & L_{11}L_{21} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 \end{bmatrix}$$

(Eq. 2.9.2)

Since the Cholesky method requires that the diagonal must be positive, the values for L_{11}, L_{21} and L_{22} are easily found.

$$A[1,1] = L_{11}^2 = 1 \implies L_{11} = 1 \qquad \text{(Eq. 2.9.3)}$$

$$A[1,2] = L_{11}L_{21} = 2 \implies L_{21} = 2 \qquad \text{(Eq. 2.9.4)}$$

$$A[2,2] = L_{21}^2 + L_{22}^2 = 8 \implies L_{22} = 2 \qquad \text{(Eq. 2.9.5)}$$

In general notation this is

$$L_{jj} = \sqrt{\mathbf{A_{jj}} - \sum_{k=1}^{j-1} L_{j,k}^2} \qquad \text{(Eq. 2.9.6)}$$

$$L_{ij} = \frac{1}{L_{jj}}(\mathbf{A_{ij}} - \sum_{k=1}^{j-1} L_{i,k}L_{j,k}) \qquad \text{for } i > j \qquad \text{(Eq. 2.9.7)}$$

Assuming a matrix $\mathbf{b} = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$, we can now find **x** by first doing forwards and backwards substitution according to Eq. 2.9.1.

$$\mathbf{Ly} = \mathbf{b} \xRightarrow{F.\ subs} \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \begin{bmatrix} b_1/L_{11} \\ \frac{(b_2 - L_{21}x_1)}{L_{22}} \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \end{bmatrix} \qquad \text{(Eq. 2.9.8)}$$

$$\mathbf{Lx} = \mathbf{y} \xRightarrow{B.\ subs} \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{bmatrix} \frac{(y_1 - L_{21}x_2)}{L_{11}} \\ y_2/L_{22} \end{bmatrix} = \begin{bmatrix} 6 \\ -\frac{3}{2} \end{bmatrix} \qquad \text{(Eq. 2.9.9)}$$

The formulas for forward and backwards substitution respectively, are

$$y_i = \frac{b_i - \sum_{k=i}^{i-1} L_{ik}y_k}{L_{ii}} \qquad \text{(Eq. 2.9.10)}$$

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} L_{ik}^T x_k}{L_{ii}^T} \qquad \text{(Eq. 2.9.11)}$$

The **x** found from using this series of forward and backwards substitution is the same as can be found by inverting **A** and multiplying with **b**.

$$\mathbf{Ax} = \mathbf{b} \implies \mathbf{A^{-1}b} = \mathbf{x} = \begin{bmatrix} 6 \\ -\frac{3}{2} \end{bmatrix} \qquad \text{(Eq. 2.9.12)}$$

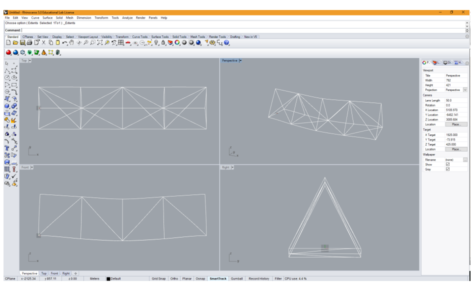
The reason this is not applicable for a global stiffness matrix is because it can be singular and thus noninvertible (Bell, 2013). Inverting the A matrix (if possible) costs $2n^3$ while, LU decomposition, a common method for solving **Ax = b**, comes at a cost of $\frac{2}{3}n^3$. The cholesky algorithm is considered to cost $\frac{1}{3}n^3$ flops for a matrix **A** of size n, so twice as quick as the LU algorithm.

Chapter 3

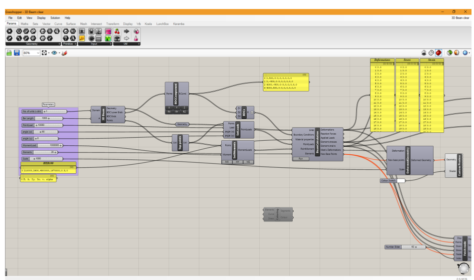
Software

This thesis is primarily focused on creating structural analysis packages for Grasshopper, which is an add-on for the computer-aided design (CAD) application *Rhinoceros 5*, often nicknamed Rhino. Rhino allows for drawing of 3D models, and makes use of non-uniform rational B-splines (NURBS) for mathematically correct drawing of curves. The user interface for Rhino can be seen on Fig. 3.1a.

Rhino has an add-on for a visual programming language called Grasshopper, see Fig. 3.1b. Grasshopper is run from within the Rhino application, and is used to build generative algorithms for geometry. These algorithms are made by pulling components onto a canvas. Components can have outputs which can subsequently be connected to other components. The process is intuitive even without prior knowledge of coding, and is very helpful for automating repeated tasks during model generation.



(a) Rhinoceros 5

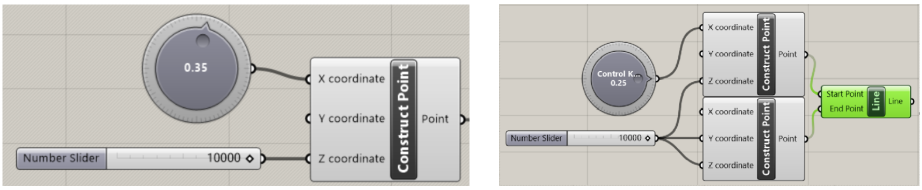


(b) Grasshopper

Figure 3.1: Parametric environment

3.1 Parametric Software

The Grasshopper add-on is a so-called ”Parametric Environment”, in which a chosen set of parameters can be used to influence the geometry to change as desired. The components can be viewed as functions, where the inputs affect the output. These parameters can be sliders, Boolean toggles, or knobs, all of which can be used to send a number or Boolean value to the components. On Fig. 3.2a, a knob and a slider is used to define the coordinates of a new point. The point component output can then be used along with another point component as inputs to a line component, as shown on Fig. 3.2b. In these two steps, an algorithm has been created for generation of a line with two nodes.



(a) Slider and knob parameters

(b) Create a line

Figure 3.2: Algorithm

The components are organized in tabs and panels on the upper part of the Grasshopper interface. For the Maths tab, the panels are Domain, Matrix, Operators, etc., as can be observed on Fig. 3.3. The Operator panels contains components for Addition, Multiplication, Smaller Than, Equality and more. Tabs are marked in blue, panels in red, and components (which can be dragged onto the canvas below) is green. Additional component packages can be downloaded and added to Grasshopper from external sources. The components created in this thesis is organized in the tab Koala, with panels for 2D Truss, 3D Truss, 3D Beam and Shell, which can be spied upon in Fig. 3.7. Each panel contains all the components related to their respective structure type.

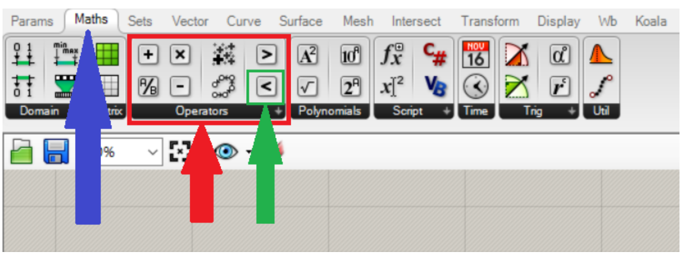


Figure 3.3: Grasshopper component organization

3.2 Installation Instructions

Grasshopper is launched from Rhino by entering the command "Grasshopper" in the Rhino command line, as shown in Fig. 3.4.

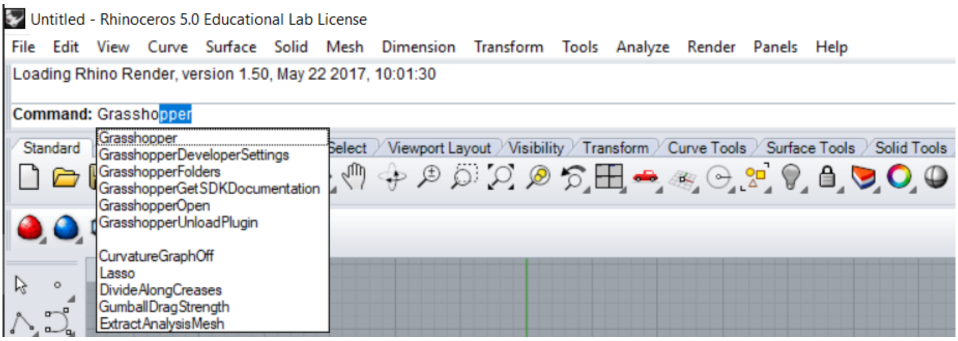


Figure 3.4: Launch Grasshopper

To add 3rd party components, go to $File \rightarrow SpecialFolders \rightarrow ComponentsFolder$ in Grasshopper, as shown on Fig. 3.5.

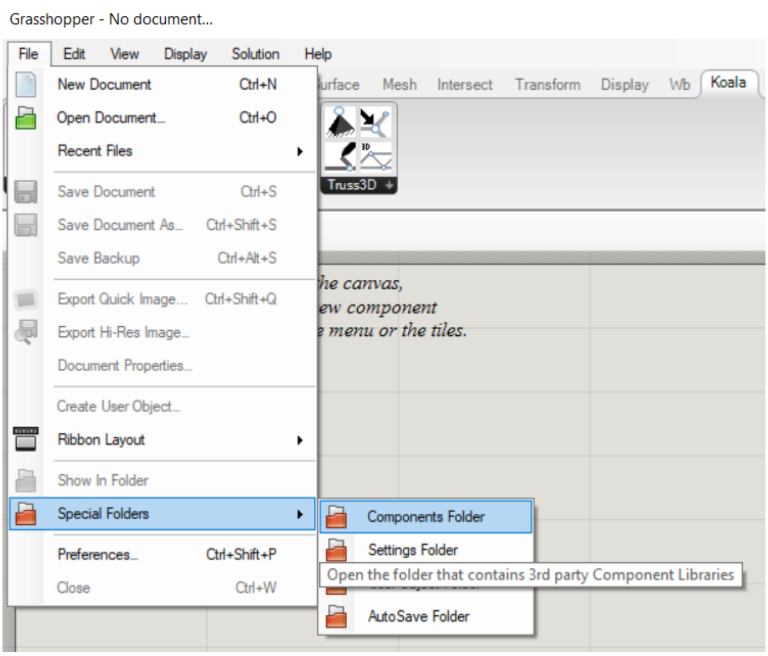


Figure 3.5: Open 3rd party Component Libraries folder

A folder containing all external libraries will open, as shown on Fig. 3.6. If no 3rd party components have been added, this folder would be empty. To add the Koala components, simply drag the whole folder (called Koala) into the libraries folder as it is shown on Fig. 3.6. Afterwards, restart Rhino and Grasshopper and the new components should be available.

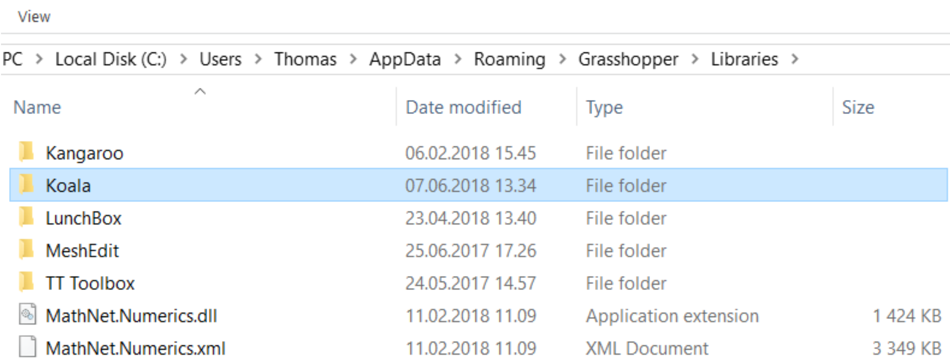


Figure 3.6: Libraries folder

After restarting Rhino and Grasshopper the Koala tab should be visible and contain all the software components created in this thesis, as shown in Fig. 3.7.

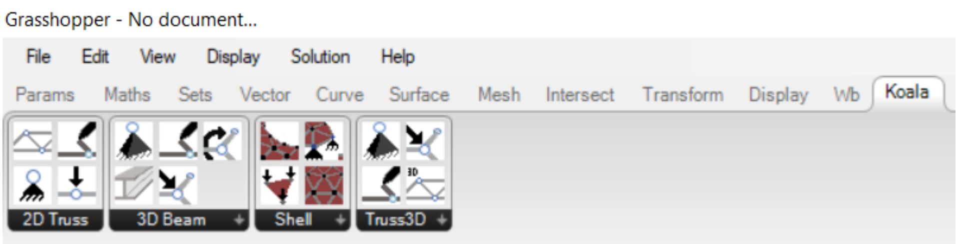


Figure 3.7: Koala tab containing all software components

Chapter 4

Truss Calculation Software

Without any experience in C# or in creation of a Finite Element Analysis (FEA) software, it was decided that creating a simple 2D truss calculation software would be a fitting introductory task. It was conjectured that the main challenges would be the ”core” of the software, which would be the solver and logic for the system of equations in matrix form. Designing the main component also introduces the finite element method (FEM) and could provide an understanding of how the method can be implemented to solve any arbitrary truss structure.

As the early work progressed it became apparent that the amount of support code needed was greater than initially assumed. Among these were the definition of boundary condition and the preparation of loads. The software was therefore dispersed among various components and methods to increase code readability and for user convenience. The need for a method to view the result also emerged as it became difficult to determine if the results were logical and consistent. It was by this reason determined that some sort of visualization of the results was in order. This functionality was placed in its own component to ease the manageability of the viewing.

When the 2D Truss calculation software were operational, the task naturally became making a 3D Truss software from the 2D version. The two software packages therefore operates very similarly, where the 3D has some extended functionality. For this reason, the 2D and 3D Truss software will be described in this chapter and an attempt will be made to point out the differences made from the 2D version to 3D. The full source codes for 2D and 3D Truss can be found in respectively Appendix A and Appendix B.

To use the software some simple relation needs to be understood, a simplified organization of the component relations is shown on Fig. 4.1.

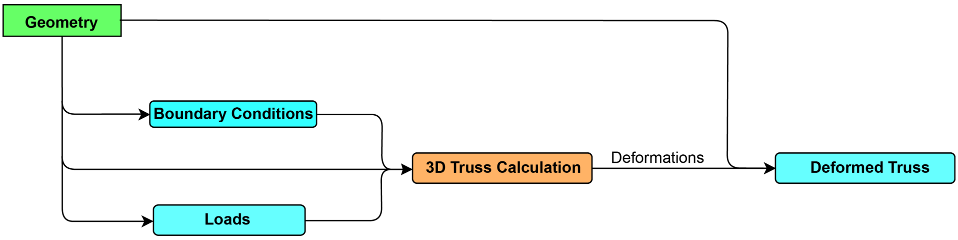


Figure 4.1: Organization of 3D Truss Components.

Where the *Geometry* represents the structure in form of lines, which in this case represents the trusses. The structures to be analyzed are built in Grasshopper, which can swiftly be adapted through parameters. The same relation pattern applies to the 2D Truss software.

4.1 Calculation Component

The inputs for the main 3D Truss components seen in Fig. 4.2 are:

Lines - The structure or geometry made with lines in the Grasshopper environment.

Boundary Conditions - The list of strings describing the support conditions for the structure, given by the *BDC Truss* component described in Ch. 4.2.1. The format is more comprehensively explained in Ch. 4.1.1.

Cross-sectional area - The cross-sectional area of the members used in the truss structure.

Material E modulus - The material parameter Young's modulus for the members of the truss structure. Describes the linear relation between stress and strain.

Loads - The loads applied to the truss structure, formatted as a list of string, given from the *SetLoads* component described in Ch. 4.2.2. The format is more thoroughly explained in Ch. 4.1.1.

The outputs from the main truss calculation component are:

Deformations - The deformation for each node in the order the nodes are found, the node order is further described in Ch. 4.1.1. The deformation are separated in respectively x, y and z direction, which gives a list three times the size the amount of unique nodes.

Reactions - Gives a list of reaction forces divided into the vector components in x, y and z direction, following the same pattern as the deformation. The reaction list also includes the applied loads in the correct positioning according to the unique load list.

Element stresses - The stresses is given as a list with the axial stress for each line in the same order they are given as input. This can be either positive or negative values for respectively tension and compression.

Element strains - The strains for each element in the same order as the lines are given as input, which is also in the same order as the stresses. Positive strain for tension and negative for compression.

When the necessary information about the geometry, boundary conditions, cross-section, material properties and loads are supplied, the various structural calculations can proceed. Initial values for E-modulus (200 GPa, assumed steel) and cross-section area (10 000 mm^2) are defined in the case either or both are unspecified.

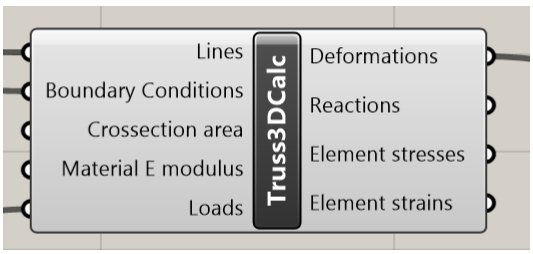


Figure 4.2: The main 3D calculation component

The main calculation component has quite a lot of tasks to perform besides the solving for deformations. There to attain a better overview of the functions, it has been separated into three parts, namely pre-processing, which is al that is done before the main calculation. The middle part is the processing which includes the main solving for the deformations, and the last part is post-processing, which will work on the results from the processing.

4.1.1 Pre-Processing

Point List

Throughout the calculation process it is important to organize all the variables. Misplaced deformations or boundary conditions in relation to the stiffness matrix will result in erroneous results. Therefore, the first step will be to create a list of all the points (i.e. nodes) from the input geometry of lines. The important thing to note about the point list is that no point occurs twice. This is done deliberately so that the point list will be the ”model order” for assembling the global stiffness matrix from the element stiffness matrices in a later procedure.

Because of lower accuracy in Grasshopper than C#, the input points are only accurate to a certain degree, and tend to have strange numbering for decimals placed after 10^{-6}. Since the Cholesky solve method outlined in Ch. 4.1.2 requires a symmetric matrix, they need to be rounded to stave off errors caused by this phenomenon. The process of creating the point list is presented in Lst. 4.1.

```
List<Point3d> points = new List<Point3d>();
foreach (Line line in geometry)
    Point3d tempFrom = new Point3d(Math.Round(line.From.X, 5),
        Math.Round(line.From.Y, 5), Math.Round(line.From.Z, 5));
    Point3d tempTo = new Point3d(Math.Round(line.To.X, 5),
        Math.Round(line.To.Y, 5), Math.Round(line.To.Z, 5));
    //adds point unless it already exists in pointlist
    if (!points.Contains(tempFrom))
        points.Add(tempFrom);
    if (!points.Contains(tempTo))
        points.Add(tempTo);
```

Listing 4.1: Method CreatePointList for 3D Truss

Having identical points occur more than once could disturb the stiffness relations in the global stiffness matrix and add more equations to the linear system, this would be unnecessary and may cause error in the computation process. The method for creating the point list will therefore skip any point if it already exists in the point list, and add it otherwise. The index for each unique point in the point list will thereafter act as the identifier for each point. It is of no consequence in which order the point occur as long as all points are unique and stays in the same order throughout the computation.

Boundary Conditions

With the arbitrary order of points established, the list of boundary conditions can be constructed. Using the *BDC Truss* component described in Ch. 4.2, the boundary conditions are given as a list of strings with the format ”x,y,z:fx,fy,fz” as shown on Fig. 4.3. The x,y and z represents the coordinates of a point in three dimensional Cartesian coordinates, given in millimetres. The field ”fx” can be interpreted as the question ”free x?”, and takes the form of an integer, 1 or 0, representing respectively *true* or *false*, the logic is similar for fy and fz.

An example of how the input is formatted for two nodes is shown in Fig. 4.3, where the nodes coordinates are as given in Eq. 4.1.1 below.

$$(x, y, z)_1 = (0, 0, 0) \qquad\qquad (x, y, z)_2 = (2000, 0, 0) \tag{Eq. 4.1.1}$$

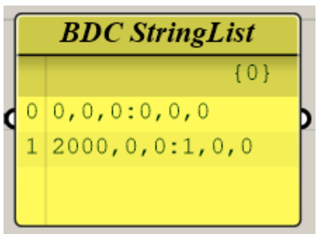


Figure 4.3: BDC string format.

Here Node 1 is clamped in x-,y-,z-direction (notice the fx,fy,fz = 0,0,0), while Node 2 is clamped in y- and z-direction, but free to move in x-direction.

The information about the points in the inputted string list is used to arrange the boundary conditions according to the order from point list. Thereafter, the boundary conditions is stored as a list with the true/false values for fx, fy and fz separated, resulting in a list with three entries for each point in the point list. For all the other points besides the boundaries, the condition is set to 1, which means it is free to move.

In the 2D Truss software, the fy value is disregarded since the calculations are only performed for two dimensions. It can be specified for testing reasons but it will be disregarded in the calculation component. Note that the y axis has been disregarded, which means 2D Truss works in the x and z axes.

Loads

Similarly to the boundary condition input, the supplied load list is a list of strings. The strings is formatted as ”x,y,z:vx,vy,vz”, where x, y and z is the coordinates of the loaded point, and vx, vy and vz is the vector components in hence x-, y- and z-direction. Each vector component has the value of the force in the respective direction, hence the complete vector contains information about direction and the force magnitude. The strings are decoded and transformed into a list of doubles, in much the same manner as for the boundary conditions. The respective force in each direction is set separately and adhering to the ordering given in the point list previously created. For 3D Truss, this results in a list thrice the length of the number of points, where all points without loads are set to zero. For 2D Truss, the length is twice the number of all points, as only two directions are considered.

Lst. 4.2 shows how the method CreateLoadList in 3D Truss parses the text input from the load component and stores them as a List of doubles.

```
for (int i = 0; i < loadtxt.Count; i++)
    string coordstr = (loadtxt[i].Split(':')[0]);
    string loadstr = (loadtxt[i].Split(':')[1]);

    string[] coordstr1 = (coordstr.Split(','));
    string[] loadstr1 = (loadstr.Split(','));

    inputLoads.Add(Math.Round(double.Parse(loadstr1[0]), 5));
    inputLoads.Add(Math.Round(double.Parse(loadstr1[1]), 5));
    inputLoads.Add(Math.Round(double.Parse(loadstr1[2]), 5));

    coordlist.Add(new Point3d(Math.Round(double.Parse(coordstr1[0]), 5),
        Math.Round(double.Parse(coordstr1[1]), 5),
        Math.Round(double.Parse(coordstr1[2]), 5)));

//place at load at correct entry in global load list
foreach (Point3d point in coordlist)
    int i = points.IndexOf(point);
    int j = coordlist.IndexOf(point);
    loads[i * 3 + 0] = inputLoads[j * 3 + 0];
    loads[i * 3 + 1] = inputLoads[j * 3 + 1];
    loads[i * 3 + 2] = inputLoads[j * 3 + 2];
```

Listing 4.2: Excerpt of method CreateLoadList for 3D Truss

Stiffness Matrices

The element stiffness matrices are established based on the geometry (List of lines), point list (List of points), Young's Modulus E and cross-sectional area A. The E-modulus and area A has been assumed to apply to all elements. The global stiffness matrix is later assembled by inserting the values from each element stiffness matrix (i.e. the element stiffness matrix of each bar) according to their node numbering, and thus connecting all the elements.

Element Stiffness Matrix

The element stiffness matrix in global coordinates $\mathbf{K}^e$ is different for 2D and 3D Truss. However, the local element stiffness matrices $\mathbf{k}^e$ are (almost) identical for both, and is similar to Eq. 2.5.59. Since the stiffness matrix from Eq. 2.5.59 is defined for one dimension, and trusses are for two and three dimensions, there is a gradual increase in the matrix size. The local element stiffness matrix for a 2D truss then becomes

$$\mathbf{k}^e_{2DTruss} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \text{(Eq. 4.1.2)}$$

The global element stiffness matrices $\mathbf{K}^e$ for 2D and 3D becomes rather different as they are multiplied by different transformation matrices. In the 2D case, the transformation matrix will be similar to the one in Eq. 2.7.13, without the rotational dof, and reads

$$\mathbf{T}_{2DTruss} = \begin{bmatrix} c & s & 0 & 0 \\ -s & c & 0 & 0 \\ 0 & 0 & c & s \\ 0 & 0 & -s & c \end{bmatrix} \qquad \text{(Eq. 4.1.3)}$$

By applying Eq. 2.7.6, this results in

$$\mathbf{K}^e_{2DTruss} = \frac{EA}{L} \begin{bmatrix} c^2 & s \cdot c & -c^2 & -s \cdot c \\ s \cdot c & s^2 & -s \cdot c & -s^2 \\ -c^2 & -s \cdot c & c^2 & s \cdot c \\ -s \cdot c & -s^2 & s \cdot c & s^2 \end{bmatrix} \qquad \text{(Eq. 4.1.4)}$$

Solving for $\mathbf{K}^e$ directly as in Eq. 4.1.4 is faster and simpler than first establishing $\mathbf{k}^e$ in local coordinates and then transforming to global coordinates. By this reason the 2D Truss component skips this transformation step and implements $\mathbf{K}^e$. For larger matrices like in 3D Truss, 3D Beam and Shell, the transformation procedure of Eq. 2.7.6 is followed instead, choosing to prioritize readability rather than optimize for time, the time usage of this process will be investigated further in later chapters. In three dimension the only difference between from the 2D local element stiffness matrix is two added rows and column of zeroes, this becomes

$$\mathbf{K}^e_{3DTruss} = \frac{EA}{L}\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{Eq. 4.1.5}$$

The transformation matrix $\mathbf{T}_{3DTruss}$ for the 3D Truss elements is found by assembling the directional cosines from Eq. 2.7.23-2.7.24 for each node and is assembled as

$$\mathbf{T}_{3DTruss} = \begin{bmatrix} C_X & C_Y & C_Z & 0 & 0 & 0 \\ C_X & C_Y & C_Z & 0 & 0 & 0 \\ C_X & C_Y & C_Z & 0 & 0 & 0 \\ 0 & 0 & 0 & C_X & C_Y & C_Z \\ 0 & 0 & 0 & C_X & C_Y & C_Z \\ 0 & 0 & 0 & C_X & C_Y & C_Z \end{bmatrix} \tag{Eq. 4.1.6}$$

The resulting $\mathbf{K}^e_{3DTruss}$ will then be calculated according to Eq. 2.7.6 for each element as the coordinates are obtained.

As an example of how the transformation matrix is used, the 2D truss element matrix from Eq. 4.1.4 is filled for element 1 on Fig. 4.4. The coordinates for the node in the figure is presented in Tab. 4.1.

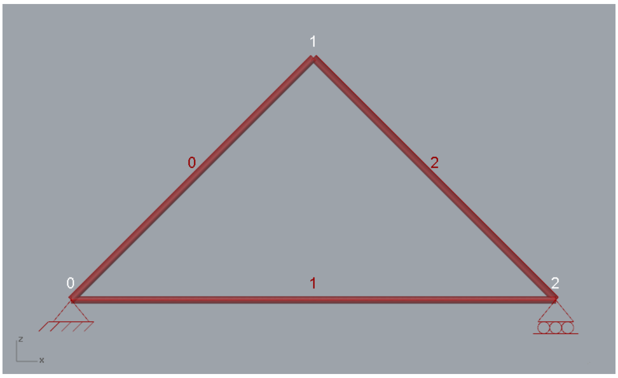


Figure 4.4: 2D Truss with red indices for elements and white indices for nodes.

Table 4.1: Example 2D Truss. Nodal coordinates for Fig. 4.4.

Node index	**X-coord**	**Y-coord**	**Z-coord**
0	0.0	0.0	0.0
1	1000.0	0.0	1000.0
2	2000.0	0.0	0.0

The angle θ is found by taking the arctangent of $\frac{\Delta z}{\Delta x}$.

$$\theta = \arctan \frac{0-0}{2000-0} \quad \text{(Eq. 4.1.7)}$$

$$= \arctan \frac{0}{2000} \quad \text{(Eq. 4.1.8)}$$

$$= 0° \quad \text{(Eq. 4.1.9)}$$

The angle is then inserted into the abbreviated sine and cosine from Eq. 2.7.12.

$$c = \cos 0° = 1 \qquad s = \sin 0° = 0 \quad \text{(Eq. 4.1.10)}$$

Material properties are set as

$$E = 210GPa \qquad A = 10000mm^2 \qquad L = 2000mm \quad \text{(Eq. 4.1.11)}$$

Inserting values from Eq. 4.1.10 to 4.1.11 into $\mathbf{K}^{e}_{2DTruss}$ from Eq. 4.1.4 results in the complete element stiffness matrix for element 1, in global coordinates.

$$\mathbf{K}^{e1}_{2DTruss} = \frac{210GPa \cdot 10000mm^2}{2000mm} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{Eq. 4.1.12}$$

$$= 10.5 \cdot 10^5 \frac{N}{mm} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{Eq. 4.1.13}$$

A remark on the example is that the global element stiffness matrix looks exactly like in Eq. 4.1.4 because it is oriented horizontally. The numbers would be ”messier” for the other diagonal elements.

Global Stiffness Matrix

The element stiffness matrix for bar element 1 is now the 4x4 matrix from Eq. 4.1.13 and next step is to add it to the global stiffness matrix. The element stiffness matrix can be divided into four 2x2 matrices: upper left corner, upper right corner, lower left corner and lower right corner. The placement of each 2x2 matrix in the element stiffness matrix is dependent on which index the start node and end node has in the point list. The four sub-matrices is illustrated in Eq. 4.1.14 with the respective node relation.

$$\mathbf{K}^{1}_{2DTruss} = 10.5 \cdot 10^5 \frac{N}{mm} \begin{array}{c} \left[\begin{array}{cc|cc} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ \hline -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \begin{array}{l} \left.\begin{array}{c} \\ \\ \end{array}\right\} \textit{Node 0} \\ \left.\begin{array}{c} \\ \\ \end{array}\right\} \textit{Node 2} \end{array} \\ \underbrace{\qquad\quad}_{\text{Node 0}} \underbrace{\qquad\quad}_{\text{Node 2}} \end{array} \tag{Eq. 4.1.14}$$

This can be automated by use of for-looping like in Lst. 4.3. Note that this double for-loop is for 3D Truss, while the 2D Truss utilized a more direct placement method as there were just a few term to place. As the element stiffness matrices grow, this process become more comprehensive and a double for-loop seemed like the organized way to accomplish this task.

```
for (int row = 0; row < K_elem.RowCount / ldof; row++)
    for (int col = 0; col < K_elem.ColumnCount / ldof; col++)
        //top left 3x3 of K-element matrix
        K_G[nIndex1 * 3 + row, nIndex1 * 3 + col] += K_elem[row, col];
        //top right 3x3 of K-element matrix
        K_G[nIndex1 * 3 + row, node2 * 3 + col] += K_elem[row, col + 3];
        //bottom left 3x3 of K-element matrix
        K_G[node2 * 3 + row, nIndex1 * 3 + col] += K_elem[row + 3, col];
        //bottom right 3x3 of K-element matrix
        K_G[node2 * 3 + row, node2 * 3 + col] += K_elem[row + 3, col + 3];
```

Listing 4.3: An automated process for $\mathbf{K}^e$ placement into $\mathbf{K}^G$ for the 3D Truss software

For the 2D Truss from Fig. 4.4, element 1 begins at node 0 and ends at node 2, this results in a placement in the global stiffness matrix as shown in Eq. 4.1.15-4.1.16.

$$\mathbf{K_1^G} = \begin{bmatrix} K^1_{0,0} & K^1_{0,1} & 0 & 0 & K^1_{0,2} & K^1_{0,3} \\ K^1_{1,0} & K^1_{1,1} & 0 & 0 & K^1_{1,2} & K^1_{1,3} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ K^1_{2,0} & K^1_{2,1} & 0 & 0 & K^1_{2,2} & K^1_{2,3} \\ K^1_{3,0} & K^1_{3,1} & 0 & 0 & K^1_{3,2} & K^1_{3,3} \end{bmatrix} \tag{Eq. 4.1.15}$$

$$= \begin{bmatrix} 10.5 \cdot 10^5 & 0 & 0 & 0 & -10.5 \cdot 10^5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -10.5 \cdot 10^5 & 0 & 0 & 0 & 10.5 \cdot 10^5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{Eq. 4.1.16}$$

An identical procedure is performed for every element, normally starting with element 0 and ending with the last element. A complete global stiffness matrix $\mathbf{K^G}$ for Fig. 4.4 will look like in Eq. 4.1.17.

The global stiffness matrix can be preallocated with a dimension of NxN by using the number of nodes multiplied with the number of local degrees of freedom (ldofs). For the 2D Truss there will always be 2 ldofs per node, in respectively x- and z-direction. While for 3D Truss there are 3 ldofs, in respectively x-, y- and z-direction. For the 2D Truss example shown in Fig. 4.4 there are three unique nodes, which gives a global stiffness matrix of 6x6 entries. A more thorough description of ldofs and gdofs can be found in Ch. 2.2.

$$\mathbf{K^G} = \begin{bmatrix} K^0_{0,0}+K^1_{0,0} & K^0_{0,1}+K^1_{0,1} & K^0_{0,2} & K^0_{0,3} & K^1_{0,2} & K^1_{0,3} \\ K^0_{1,0}+K^1_{1,0} & K^0_{1,1}+K^1_{1,1} & K^0_{1,2} & K^0_{1,3} & K^1_{1,2} & K^1_{1,3} \\ K^0_{2,0} & K^0_{2,1} & K^0_{2,2}+K^2_{0,0} & K^0_{2,3}+K^2_{0,1} & K^2_{0,2} & K^2_{0,3} \\ K^0_{3,0} & K^0_{3,1} & K^0_{3,2}+K^2_{1,0} & K^0_{3,3}+K^2_{1,1} & K^2_{1,2} & K^2_{1,3} \\ K^1_{2,0} & K^1_{2,1} & K^2_{2,0} & K^2_{2,1} & K^1_{2,2}+K^2_{2,2} & K^1_{2,3}+K^2_{2,3} \\ K^1_{3,0} & K^1_{3,1} & K^2_{3,0} & K^2_{3,1} & K^1_{3,2}+K^2_{3,2} & K^1_{3,3}+K^2_{3,3} \end{bmatrix}$$

$$= \begin{bmatrix} 17.9 & 7.4 & -7.4 & -7.4 & -10.5 & 0 \\ 7.4 & 7.4 & -7.4 & -7.4 & 0 & 0 \\ -7.4 & -7.4 & 14.8 & 0 & -7.4 & 7.4 \\ -7.4 & -7.4 & 0 & 14.8 & 7.4 & -7.4 \\ -10.5 & 0 & -7.4 & 7.4 & 17.9 & -7.4 \\ 0 & 0 & 7.4 & -7.4 & -7.4 & 7.4 \end{bmatrix} \cdot 10^5 \frac{N}{mm} \qquad \text{(Eq. 4.1.17)}$$

When assembled the constant global stiffness matrix can describe the linear static behaviour of the structure, by providing a relation between forces and deformations.

Reduced Global Stiffness Matrix and Reduced Load List

After the global stiffness matrix has been established, the reduced global stiffness matrix ($\mathbf{K}_r^G$-matrix) must be constructed. In order to create the $\mathbf{K}_r^G$-matrix, the clamped boundary conditions are removed. It is also necessary to reduce the load list equivalently so that it can be used as the right-hand-side (RHS) of the equation when solving for displacements. The process of solving is more thoroughly explained in Ch. 2.3 and Ch. 2.9.

For every entry in the boundary list containing zeros, the corresponding index for rows and columns in $\mathbf{K}^G$ and load list is removed, as illustrated in Eq. 4.1.18. Numbers in black will remain in the new list, while the rest (grey) numbers are removed.

The RHS of Eq. 4.1.18 will be the reaction and loading forces, where the reaction forces at this time are still unknowns and are therefore set as zeroes in the software. The equations with reaction forces as the RHS can not be solved without more information and is therefore not taken into the reduced stiffness matrix. The equations involving reaction forces however, does not relate to any deformations as they relates to clamped dofs, which are not movable. They can therefore be removed and the system can be solved without any complications.

$$\underbrace{\left[\begin{array}{c} 0 \\ 0 \\ \hline 1 \\ 1 \\ 1 \\ \hline 0 \end{array}\right]}_{\text{BDC list}} \rightarrow \underbrace{10^5 \left[\begin{array}{cc|ccc|c} 17.9 & 7.4 & -7.4 & -7.4 & -10.5 & 0 \\ 7.4 & 7.4 & -7.4 & -7.4 & 0 & 0 \\ \hline -7.4 & -7.4 & 14.8 & 0 & -7.4 & 7.4 \\ -7.4 & -7.4 & 0 & 14.8 & 7.4 & -7.4 \\ -10.5 & 0 & -7.4 & 7.4 & 17.9 & -7.4 \\ \hline 0 & 0 & 7.4 & -7.4 & -7.4 & 7.4 \end{array}\right]}_{\mathbf{K}^G} \underbrace{\left[\begin{array}{c} u_0 \\ v_0 \\ \hline u_1 \\ v_1 \\ u_2 \\ \hline v_2 \end{array}\right]}_{\mathbf{u}} = \underbrace{\left[\begin{array}{c} Reac. \\ Reac. \\ \hline Load \\ Load \\ Load \\ \hline Reac. \end{array}\right]}_{\text{load list}}$$

(Eq. 4.1.18)

The resulting $\mathbf{K}_r^G$-matrix becomes as shown in Eq. 4.1.19, along with the reduced load list.

$$\underbrace{10^5 \begin{bmatrix} 14.8 & 0 & -7.4 \\ 0 & 14.8 & 7.4 \\ -7.4 & 7.4 & 17.9 \end{bmatrix}}_{\mathbf{K}_r^G} \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}}_{\text{BDC list}} = \underbrace{\begin{bmatrix} Load \\ Load \\ Load \end{bmatrix}}_{\text{reduced load list}} \quad \text{(Eq. 4.1.19)}$$

While the 3D Truss makes use of inbuilt functions for lists and matrices to remove rows and columns by index, the 2D Truss instead builds the reduced stiffness matrix (and reduced load list) from scratch by adding all the entries which relates to free dof in the $\mathbf{K}^G$ matrix and load list. As will be explained in Ch. 5.1.1, the method for 2D Truss is actually superior to the ”improved” 3D Truss method in terms of runtime.

The difference in the algorithms for reducing the global stiffness matrix in 2D Truss and 3D Truss can be seen from respectively Lst. 4.4 and Lst. 4.5. Notice how the 3D Truss reducing method seem simpler because of the methods *RemoveRow* and *RemoveColumn*. But in fact it is noticeable slower than the 2D Truss method when the matrices grow quite large.

```
int dofs_red = points.Count * 2 - (bdc_value.Count - bdc_value.Sum());
double[,] K_redu = new double[dofs_red, dofs_red];
List<double> load_redu = new List<double>();
List<int> bdc_red = new List<int>();
int m = 0;
for (int i = 0; i < K_tot.GetLength(0); i++)
    if (bdc_value[i] == 1)
        int n = 0;
        for (int j = 0; j < K_tot.GetLength(1); j++)
            if (bdc_value[j] == 1)
                K_redu[m, n] = K_tot[i, j];
                n++;
        load_redu.Add(load[i]);
        m++;
```

Listing 4.4: CreateReducedGlobalStiffnessMatrix method for 2D Truss

```
K_red = Matrix<double>.Build.SparseOfMatrix(K);
List<double> load_redu = new List<double>(load);
for (int i = 0, j = 0; i < load.Count; i++)
    if (bdc_value[i] == 0)
        K_red = K_red.RemoveRow(i - j);
        K_red = K_red.RemoveColumn(i - j);
        load_redu.RemoveAt(i - j);
        j++;
```

Listing 4.5: CreateReducedGlobalStiffnessMatrix method for 3D Truss

4.1.2 Processing by Cholesky-Banachiewicz Algorithm

After reducing the global stiffness matrix and load list, Eq. 2.3.1 can be solved for displacements. Explanation of Cholesky Decomposition and reasons for choosing it over other methods can be found in Ch. 2.9 and analysis of some solvers are performed in Ch. 5.3.1.

This part of the software is where the 2D and 3D Truss diverges more drastically. While 2D Truss contains a self-written algorithm for solving with the Cholesky method and matrix multiplication, the 3D Truss instead utilizes the Math.NET Numerics package (Math.NET, 2018). In essence, Math.NET opens for use of readily built and optimized methods and classes. Matrix and vector classes are introduced and matrix operations can be performed by pre-built solver methods. The Cholesky algorithm as well as forwards and backwards substitution can all be replaced with Math.NET functions. A runtime comparison between our Cholesky and the Math.NET Cholesky is shown in Chapter 5.3.1.

Cholesky Decomposition and Restoration of Displacement list, 2D Truss

To construct $\mathbf{L}$ and $\mathbf{L_T}$ from Eq. 2.9.1 a double for-loop goes through the whole $\mathbf{K_r^G}$-matrix and calculates Eq. 2.9.6 for diagonal entries and Eq. 2.9.7 for remaining entries. The sums are stored in a temporary variable, L_{sum}. Note that diagonal and general entries are different.

$$L^D_{sum} = L^D_{sum} + L^2_{i,k} \qquad L^G_{sum} = L^G_{sum} + L_{i,k}L_{j,k} \qquad \text{(Eq. 4.1.20)}$$

The implementation is presented in LST. 4.6

```
for (int i = 0; i < KGr.GetLength(0); i++)
    for (int j = 0; j <= i; j++)
        double L_sum = 0;
        if (i == j)
            for (int k = 0; k < j; k++)
                L_sum += L[i, k] * L[i, k];
            L[i, i] = Math.Sqrt(KGr[i, j] - L_sum);
            L_T[i, i] = L[i, i];
        else
            for (int k = 0; k < j; k++)
                L_sum += L[i, k] * L[j, k];
            L[i, j] = (1 / L[j, j]) * (KGr[i, j] - L_sum);
            L_T[j, i] = L[i, j];
```

Listing 4.6: Construction of L and $\mathbf{L^T}$

After $\mathbf{L}$ and $\mathbf{L^T}$ has been constructed they can be forward substituted for given loads. As per the procedure outlined in Ch. 2.9, this is done by solving for Eq. 2.9.10, and is shown in Lst. 4.7

```
for (int i = 0; i < L.GetLength(1); i++)
    double L_prev = 0;
    for (int j = 0; j < i; j++)
        L_prev += L[i, j] * y[j];
    y.Add((load1[i] - L_prev) / L[i, i]);
```

Listing 4.7: Forwards substitution

When this is completed, the deformations can be found by backwards substitution as in Eq. 2.9.11. The algorithm created for this is shown in Lst. 4.8.

```
for (int i = L_T.GetLength(1) - 1; i > -1; i--)
    double L_prev = 0;
    for (int j = L_T.GetLength(1) - 1; j > i; j--)
        L_prev += L_T[i, j] * x[j];
    x[i] = ((y[i] - L_prev) / L_T[i, i]);
```

Listing 4.8: Backwards substitution

After solving $\mathbf{K}_r^G$ for deformations by the Cholesky algorithm, the removed entries (from reducing the load list) are restored by adding zeros at indices where displacements are clamped, and entries from the reduced deformations list where they free. This process is implemented as shown in Lst. 4.9.

```
List<double> def = new List<double>();
int index = 0;
for (int i = 0; i < bdc_value.Count; i++)
    if (bdc_value[i] == 0)
        def.Add(0);
    else
        def.Add(deformations_red[index]);
        index += 1;
```

Listing 4.9: Restore deformation vector, 2D Truss

Cholesky Decomposition and Restoration of Displacement list, 3D Truss

In contrast to 2D the 3D Truss software utilizes the Math.NET Cholesky solver, then preallocates the complete deformation vector with zeros and repopulates it with the calculated deformation values. The restoration process in this case becomes as shown in Lst. 4.10.

```
Vector<double> def_red = KGr.Cholesky().Solve(load_r);
Vector<double> def = Vector<double>.Build.Dense(bdc_value.Count);
for (int i = 0, j = 0; i < bdc_value.Count; i++)
    if (bdc_value[i] == 1)
        def[i] = def_red[j];
        j++;
```

Listing 4.10: Solve and restore deformation vector, 3D Truss

4.1.3 Post-Processing

Reaction Forces

Reaction forces are found by postmultiplying the complete global stiffness matrix with the complete deformation vector as per Eq. 2.3.1. The 3D Truss software solves this by matrix multiplication methods provided by Math.NET, while Lst. 4.11 shows how the reaction forces are found in 2D Truss.

```
for (int i = 0; i < K_tot.GetLength(1); i++)
    if (bdc_value[i] == 0)
        double R_temp = 0;
        for (int j = 0; j < K_tot.GetLength(0); j++)
            R_temp += K_tot[i, j] * def[j];
        R.Add(Math.Round(R_temp, 2));
    else {R.Add(0);}
```

Listing 4.11: Method CalculateReactionforces in 2D Truss

Note that the 2D Truss method only finds the reaction forces, while the 3D Truss method will result in a reaction list which also contains the applied loads.

Internal Strains and Stresses

Strain is the difference in length divided by original length, as defined in Eq. 2.5.48. By looping over each bar in the truss structure and calculating deformed and undeformed length, the strains are easily obtained. Stresses are then found by Hooke's Law, calculated in accordance with Eq. 2.5.47. This is a very simple and basic prosess and will therefore not be shown.

Outputs

After calculating the reaction forces, nodal deformations, in addition to internal strains and stresses, the outputs are produced in forms of lists of numbers. The list of nodal deformations can be given to the *Deformed Geometry* component if the user wishes to see the deformed structure.

4.2 Support Components

The support components, which are illustrated in blue on Fig. 4.1, are used to administer the boundary conditions and loads in the correct format for the main calculation component. The third support component, *Deformed Geometry*, draws the deformed geometry based on the output from the Truss Calculation component. The relation between the support component and the main component as it appears in Grasshopper is shown on Fig. 4.5.

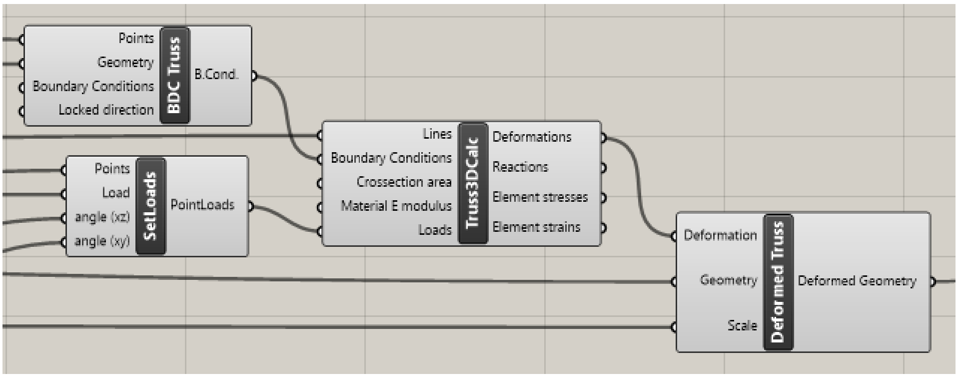


Figure 4.5: Relation between support components and main component.

4.2.1 The BDC Truss Component

The boundary conditions (BDCs) are created in the *BDC Truss* component. The inputs needed to define the boundary conditions are:

Points - Nodal coordinates (x, y, z) of each node containing a boundary condition. These are given in the form of lists of Point3d objects (which is a Rhinocommons data type). Since it was known that 2D would be extended to 3D eventually, the coordinate logic was equipped to handle 3D from the start. In some cases it can also be useful to override all other boundary conditions and clamp the entire structure in one direction (especially for 2D Truss). In order to accommodate this, the component needs the coordinates of all the nodes in the structure. By inputting the geometry, all points in the structure can retrieved and set to clamped at request.

Boundary Conditions - Since a truss structure has three translational dofs per node, each inputted node must (theoretically) be accompanied by three dof specifications. The boundary conditions for the restrained points are given as a list of 1s (free) and 0s (clamped), where every three numbers correspond to one node (e.g. 0,1,0). The component allows the user to set fewer numbers than $points * 3$, with the stipulation that all remaining points will be set according to the last three numbers in the list. This also means that three numbers can be given and they will be applied to all specified BDC points.

The resulting output is given as a list of text strings with the coordinates followed by the conditions. The conditions will be formatted as "fx,fy,fz". The complete output string format looks like "x,y,z:fx,fy,fz". The reason for this format was explained in Ch. 4.1.1.

4.2.2 The SetLoads Component

Point loads are generated through the *SetLoads* component. As input, the component requires:

Points - The points, also as Point3d objects, to which load shall be applied.

Load - The load magnitude(s) in Newtons, this can be given as one load which is to be applied to all points, or a list of loads to apply to each corresponding point in the *Points* input. If the point list happen to be longer than the load list, the last load entry will be applied to the remaining points.

The angle(s) are not a necessary input as they are preset to give the loads in negative z direction, and are formulated in degrees. While the 3D Truss works with angles both in XY-plane ($\measuredangle_{xy}$) and from the XY-plane to the load vector ($\measuredangle_{xz}$), the 2D Truss has been restricted to only the XZ-plane (which is equivalent to $\measuredangle_{xz}$ when $\measuredangle_{xy}$ = 0). The force is directed towards the node, as can be observed on Fig. 4.6. By default, $\measuredangle_{xz}$ is set to 90 degrees and $\measuredangle_{xy}$ to 0 degrees.

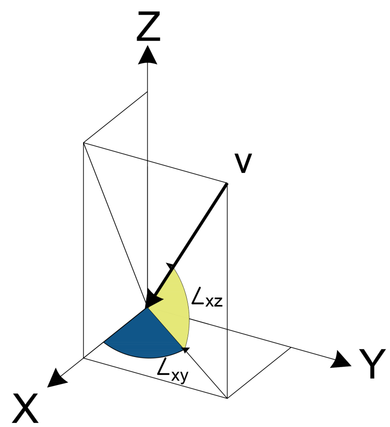


Figure 4.6: Angles for load vector **v**

The calculated load vectors are then decomposed to into the direction vectors v_x, v_y and v_z as in Eq. 4.2.1-4.2.3.

$$v_x = \cos\left(\measuredangle_{xz} \cdot \frac{\pi}{180}\right) \cos\left(\measuredangle_{xy} \cdot \frac{\pi}{180}\right) \tag{Eq. 4.2.1}$$

$$v_y = \cos\left(\measuredangle_{xz} \cdot \frac{\pi}{180}\right) \sin\left(\measuredangle_{xy} \cdot \frac{\pi}{180}\right) \tag{Eq. 4.2.2}$$

$$v_z = \sin\left(\measuredangle_{xz} \cdot \frac{\pi}{180}\right) \tag{Eq. 4.2.3}$$

Similarly to the boundary conditions, the load is outputted as a list of text strings. These are formatted as "x,y,z:vx,vy,vz", where x,y,z represents the point coordinates and vx,vy,vz represents the decomposed vectors along each axis.

4.2.3 The Deformed Truss Component

Rather than placing more strain on the main calculation component than necessary, it would be useful to have a separate component for generation of deformed geometry. Visualization of the displacements can be very useful for spotting errors and understanding the structural response to given loading and boundary conditions. This new support component, *Deformed Truss*, takes in the deformations calculated from the main component, as well as original geometry and a scale factor. A deformation scale of 0 generates a geometry similar to the initial geometry, a scale of 1 shows the actual deformed geometry, and a scale of 1000 shows a deformed geometry with a thousand times larger displacements than the actual values. The components basic logic can be seen in Lst. 4.12

```
int index = 0;
//loops through all points and scales x-, y- and z-dir
foreach (Point3d point in points)
    //fetch global x,y,z placement of point
    double x = point.X;
    double y = point.Y;
    double z = point.Z;
    //scales x and z according to scale input
    defPoints.Add(new Point3d(x + scale * def[index], y + scale *
        def[index + 1], z + scale * def[index + 2]));
    index += 3;
```

Listing 4.12: Excerpt of 3D Truss deformed geometry component

4.3 Analysis

In this chapter, the accuracy of 2D and 3D truss software are compared with Robot Structural Analysis, as well as analytical solutions where easily obtainable. Both 2D and 3D has been tested for a small range of structures to ensure that the deformation patterns looks as one would expect.

The first structure is a single bar of 2.0 m hinged in the left node and with a movable hinge on the right, see Fig. 4.7. An axial force of 1000 kN is applied on the right node.

$$A = 10000 \text{ [mm}^2\text{]} \qquad E = 210000 \text{ [MPa]}$$

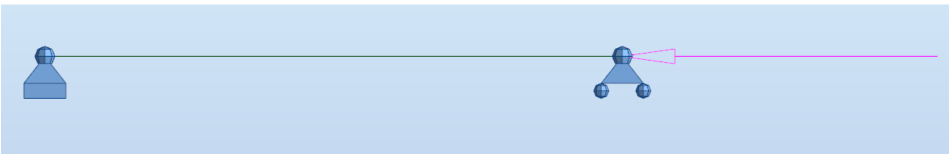


Figure 4.7: 2D Truss in Robot

The results can be viewed in Tab. 4.2, the 2D Truss and 3D Truss software bundles are referred to simply as 2D Truss and 3D Truss, while the solution from Robot Structural Analysis is referred to as Robot.

Table 4.2: Axial compression of single bar

Solution	**Displacement** [mm]	**Strain**	**Stress** [MPa]
Analytical	-0.952381	-0.000476	-100
Robot	-0.952381	N/A	-100
2D Truss	-0.952381	-0.000476	-100
3D Truss	-0.952381	-0.000476	-100

As the 2D and 3D Truss software seems to work similarly as they gave the expected same result, and all solutions were exactly alike. This gives a good indication that the implemented stiffness relations and basic coding is working as intended.

Another interesting check would be a more complex structure of three dimensions, which mean the 2D Truss will not be included. The analytical solution would also be quite hard to attain, and has not been prioritized as Robot is considered accurate enough for this test.

The structure in question can be seen in Fig. 4.8, where each of the five loads is set to 10 kN, which gives a total og 50 kN distributed among the five top middle nodes. The two boundary conditions on the left side in Fig. 4.8 has been set to pinned, while the two supports on the right side is set to roller-supports, allowing for deformation in x direction.

$$A = 1836 \text{ [mm}^2\text{] [m]} \qquad\qquad E = 210000 \text{ [MPa]}$$

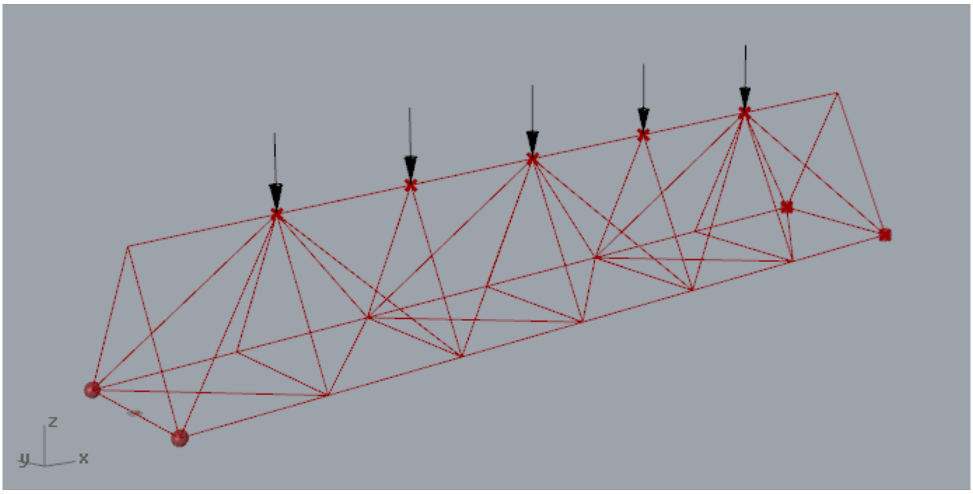


Figure 4.8: 3D Truss in Grasshopper/Rhino

The deformed structure can be seen in Fig. 4.9 as the white structure. It has been colored white so it would be easier to see the deformation, the scale for deformation is set to 300 to give a clear view of the deformation.

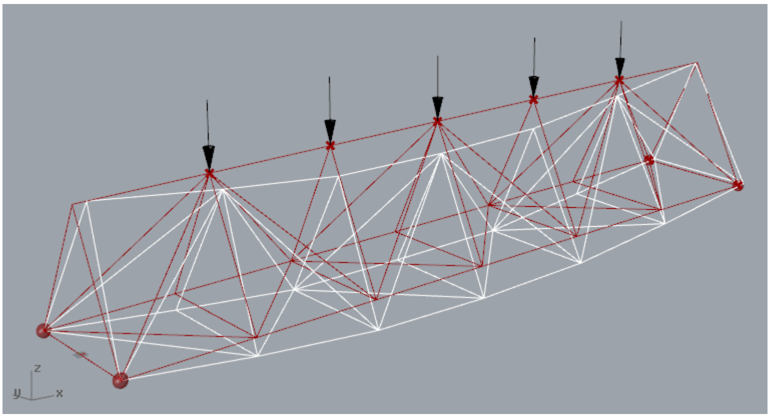


Figure 4.9: Deformed 3D Truss in Grasshopper/Rhino

The deformed structure looks very similar in Robot, which indicates that the structural system responded in a similar way. The deformation values from Robot and Grasshopper can be viewed in Tab. 4.3 below.

Table 4.3: Highest deformation and stress for truss in Fig. 4.9

Solution	**Displacement** [mm]	**Stress** [MPa]
Robot	-1.7620	-21.7865
3D Truss	-1.762047	-21.7861

As can be seen, Robot gave very similar results to our truss software. Which could qualify the truss software as ”pretty exact” as it shows. For confirmation of the good results the same test were repeated with Area A = 374 mm^2, which gave the results in Tab. 4.4.

Table 4.4: Highest deformation and stress for truss with A = 374 mm^2

Solution	**Displacement** [mm]	**Stress** [MPa]
Robot	-8.6501	-106.9519
3D Truss	-8.650051	-106.9421

Which again shows some very close values.

4.4 Discussion

The organization of the components was found to be quite easy to use when separated into several parts, the support and loading preparations gave a good way to check the loads and boundary condition before they were sent to the main component. The separated components also gave and opportunity to use several components to assemble different loading and boundary conditions and merge them, or swiftly switch between them. This gave a practical and fast way to check different supporting and loading conditions.

The implementation of the 2D and 3D Truss softwares gave a quite good introduction to both simple FEM principles and how to implement them as software. Several methods was found to improve the coding, and will hopefully give an advantage when moving to the beam structures. The basic principles of finite element analysis for bars seemed simpler to understand before attempting to implement them in an arbitrary software for all kinds of structures. The process was not particularly difficult, but it may take some time to perfect all the details to attain a realistic solution.

Some insight into the finite element analysis was attained even though the element is the simplest possible. The creation of our own solver gave quite a good insight into the mathematics behind the ”core” of the calculation. As was conjectured the ”core and it’s logic” was surprisingly not the hardest part of the software but actually rather interesting, especially as it proves to be the most time consuming process, which will be more thoroughly examined in Ch. 5.3. The relatively more difficult part was the total organization of all the data inside the main calculation component. This seems like an important piece experience as the number of dofs and elements may be substantially higher in the beam and shell softwares.

One unforeseen issue was the organization of the elements and dofs as the number of element became higher. The total amounts of dofs made it severely difficult to get an good overview of the results, and the *Deformed Truss* had to be made to be able to assert if the solution was realistic or not. It could also be a valuable tool to have the component show color-maps for stresses in the bars, so that critical point easily can be identified visually. This is however an easy task to do for an experienced Grasshopper user.

It also became clear that one small error could make a huge impact on the solution, and the software therefore should be tested extensively before being classified as finished. The process of finding and fixing bugs in the code also proved to be a larger part of this project than anticipated.

The results in the analysis between Robot and our 3D Truss software was quite interesting. They may show just how simple it is to make an easy and ”less advanced” calculation software. As explained in Ch. 2, our software is based on some assumptions which simplifies much of the calculations, and yet were the results very similar.

An interesting expansion to the software could be to expand the possibilities for different materials and cross-sections for different bars in the same structure.

4.5 Truss Summary

The created 3D Truss software establishes the foundation for further development as the organization and structure was found to be advantageous for checking results and locating smaller and larger errors. The Truss softwares are very simple in theory and not extensively hard to implement with some knowledge about finite element analysis, the field of mechanics and programming.

There were made some mistakes that may prove useful for later advancement to more complex finite elements as beams. One of these mistakes were the method for reducing the global stiffness matrix used in 2D Truss versus the one used in 3D Truss. The ”upgraded version” proves to be slower as it perform more unnecessary operations, and has given some valuable experience for further work.

The 2D and 3D software packages worked very well compared to Robot, and gave relatively very close results. The software can there be said to work properly and as intended, which also means that they are quite simplistic.

The introduction of Math.NET Numerics greatly improved the solving process as the Math.NET package are both more adaptive to problems and more general in solving methods. Another aspect of the Math.NET package, namely running time, will be more thoroughly explored in Ch. 5.3, as the structures becomes more complex and time consuming to calculate.

Chapter 5

3D Beam Calculation Software

Similarly to the Truss software described in Ch. 4, the 3D Beam software is comprised of a main component for calculation and three pre-processing support components as shown on Fig. 5.1. In addition, there is of course a post-processing component for visualization of the deformed geometry. The full source code for 3D Beam can be found in Appendix C.

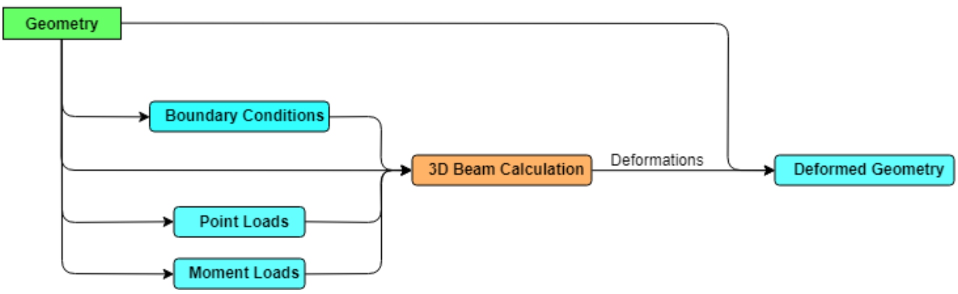


Figure 5.1: Organization of 3D Beam Components.

The main differences between Truss and Beam software lies in the generation of the element stiffness matrices and the utilization of shape functions. Multiple toggles have been added to the graphics of the components so that they are easier to use. This is more thoroughly explained in each individual component's section.

5.1 Calculation Component

The main calculation component *BeamCalculation* is shown in Fig. 5.2. Input geometry must be given as lines, similarly to the Truss software. The boundary conditions and loads are also given in the same format as for Truss. In addition, this component takes in moment loads which are formatted in the same manner as point loads, except with moment force instead of decomposed force vectors.

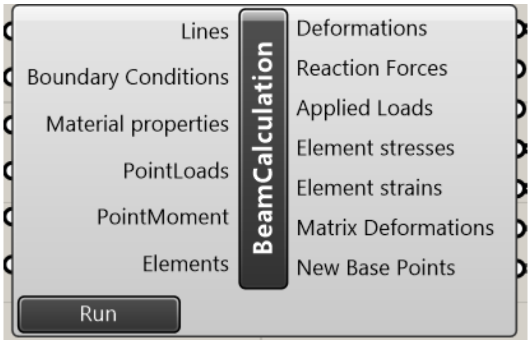


Figure 5.2: Main calculation component for 3D Beam

Material properties are given as a string of numbers, formatted as ”E, A, Iy, Iz, v, alpha”. It must include the second moment of area about both beam axes (I_z and I_y), in addition to Poisson’s ratio (v) and rotation about the local x-axis, alpha. The input named ”Elements” refers the number of sub-elements the beam elements should be divided in, for calculation of nodes within elements and better preview of the bending of the beam elements.

The outputs for deformations, reaction forces, applied loads, stresses and strains are lists of doubles, following the ordering of nodes as they are given as input (Lines input). The outputs called Matrix Deformations and New Base Points are to be handed over to the *Deformed Geometry* component, and is further explained in Ch. 5.1.3 and Ch. 5.2.3.

A simplified workflow for the algorithm inside the *BeamCalculation* component is shown on Fig. 5.3. The component is roughly divided into pre-processing, processing and post-processing to simplify the organization of the internal programming. Note that calculations performed inside elements could be interpreted as both part of processing and post-processing.

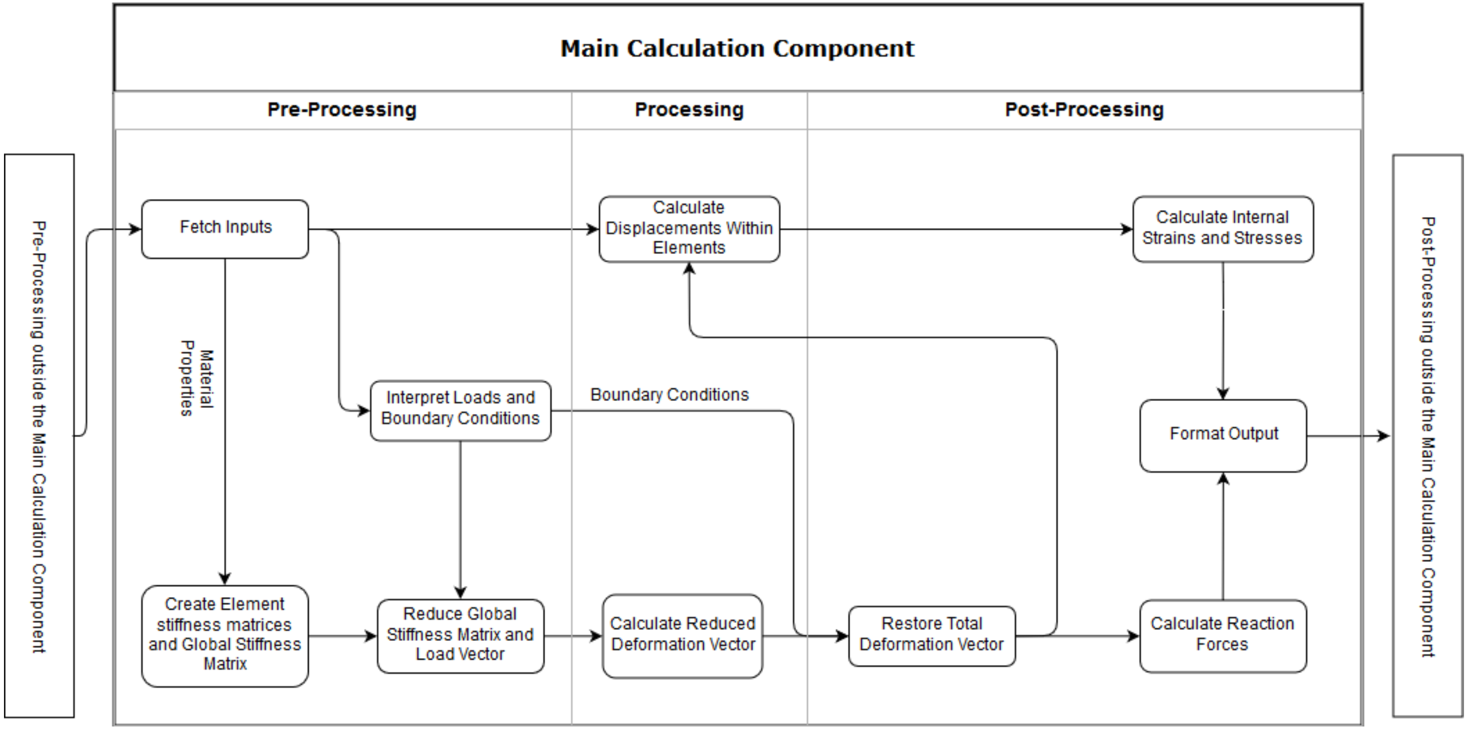


Figure 5.3: Simplified workflow of the main component in Beam

5.1.1 Pre-Processing

The creation of a point list, BDC list and load list is done the same way as for 3D truss, see Ch. 4.1.1.

Global Stiffness Matrix

The element stiffness matrices $\mathbf{k^e}$ must be constructed before the stiffnesses can be assembled into $\mathbf{K^G}$. $\mathbf{k^e}$ is built according to the procedure shown in Ch. 2.5.3. The main difference between the Truss and Beam software is obviously the expansion to rotational stiffness. The complete 12-dof $\mathbf{k^e}$-matrix that is used for the Beam component is shown in Eq. 2.5.67.

The element stiffness matrix is then transformed from local to global coordinates by Eq. 2.7.6. Since the transformation matrix **tf** utilizes directional cosines, only the coordinates for start and end node is needed. The procedure outlined in Ch. 2.7 is used to construct **tf**. The resulting 3x3 matrix is stacked diagonally like in Eq. 2.7.14 to form **T** as needed.

After $\mathbf{k^e}$ has been transformed to $\mathbf{K^e}$, the last step is to place the stiffnesses at their correct entries in the global stiffness matrix $\mathbf{K^G}$. This algorithm is similar to the one for 3D Truss.

Reduce

Drastic code improvements are made on the method to create the reduced global stiffness matrix in comparison to Lst. 4.5 for 3D Truss. The 3D Truss method generated a completely new matrix for every row and column removed, resulting in needlessly long runtime. The new algorithm in Lst. 5.1 is more efficient and preallocates $\mathbf{K^G_r}$ and $\mathbf{load_r}$, then fills them with the correct values by use of double for-loops while checking for unclamped boundary conditions. Initially, the algorithm looped through the whole matrix while creating $\mathbf{K^G_r}$. This was improved to exploit $\mathbf{K^G}$'s symmetric property after analysis done for shell showed that the algorithm was surprisingly slow. More on the new algorithm can be found in Ch. 6.1.1.

```
int oldRC = load.Count;
int newRC = Convert.ToInt16(bdc_value.Sum());
KGr = Matrix<double>.Build.Dense(newRC, newRC);
load_r = Vector<double>.Build.Dense(newRC, 0);
for (int i = 0, ii = 0; i < oldRC; i++)
    if (bdc_value[i] == 1)  //is bdc_value in row i free?
        for (int j = 0, jj = 0; j < oldRC; j++)
           if (bdc_value[j] == 1)  //is bdc_value in col j free?
                //if yes, then add to new K
                KGr[i - ii, j - jj] = K[i, j];
                KGr[j - jj, i - ii] = K[i, j];
            else //if not, remember to skip 1 column when adding next time
                jj++;
        load_r[i - ii] = load[i]; //add load to reduced list
    else    //if not, remember to skip 1 row when adding next time
        ii++;
return KGr;
```

Listing 5.1: ReduceStiffnessMatrix method for 3D Beam

5.1.2 Processing

Displacements

The nodal displacements are found by Cholesky Decomposition. Same as for 3D Truss, this is accomplished by use of a Math.NET function. More on this in Ch. 2.9 and Lst. 4.10.

```
Vector<double> def_red = KGr.Cholesky().Solve(load_red);
```

Displacements Within Element

The approximate displacements within each element can be found by Eq. 2.5.1. The procedure requires each element's displacement vector $\mathbf{\Delta^e}$, as well as its length L, transformation matrix **tf**, and requested number of sub-elements n. The displacement fields are similar to **N** and **dN** from Eq. 2.5.42-2.5.43.

The code variable for the element displacement vector $\mathbf{\Delta^e}$ is u. After retrieving u from the global displacement vector, it is transformed to local coordinates by Eq. 2.5.2.

The displacement fields are constructed in a method called $DisplacementField\_NB$. There, the shape functions from Eq. 2.5.29-2.5.34 and Eq. 2.5.35-2.5.40 are calculated for the given x and L, and subsequently used to construct the fields. The resulting **N** and **dN** are postmultiplied by the displacement vector **u** to get the nodal displacements, see Eq. 2.5.1. These nodal displacements are then transformed to global coordinates, see Eq. 2.5.3.

```
foreach element in elements
    L = DistanceBetween(endpoint1, endpoint2)
    Get nodal displacements of endpoints, u_g
    u_l = T^T * u_g
    x = 0

    foreach node in subelements
        N and dN = DisplacementField_NB(x, L)
        [u_x, u_y, u_z, θ_x]_l^n = N * u_l
        [−, θ_z, θ_y, −]_l^n = dN * u_l

        [u_x, u_y, u_z, θ_x, θ_y, θ_z]_g^n = T^T * [u_x, u_y, u_z, θ_x, θ_y, θ_z]_l^n
        x = x + L / n
```

Listing 5.2: Pseudocode for interpolated displacements

5.1.3 Post-Processing

Reaction Forces

Same as for 3D truss, the general forces are found by postmultiplying $\mathbf{K}^{\mathrm{G}}$ with the deformations. Since the applied loads are stored in a separate list, a list containing only the reaction forces can be found by subtracting the applied loads from the general forces list.

Strains and Stresses

The shape functions can also be used to find the strains within each element, as explained in Ch. 2.5.2. By similar procedure as was performed in Ch. 5.1.2, the displacement fields **dN** and **ddN** are calculated. By reusing the nodal displacement vector **u** for each element, the strains per node is found. Stresses are calculated by the relation $\sigma = E\varepsilon$.

Format Output

Since Grasshopper cannot operate on Math.NET matrices, deformations, strains and stresses are converted to double[] lists. To make it easier for the *Deformed Geometry* component to calculate deformed geometry, the deformations are also given as output in their original Math.NET matrix form. The *Deformed Geomtry* component utilizes Math.NET, therefore it has no problems interpreting the matrix. Lastly, nodal coordinates of the sub-elements are found by interpolating the original geometry and sent along as output for the *Deformed Geometry* component to apply the displacements to.

```
foreach Line line in lines
    double[] t = LinSpace(0, 1, n + 1);
    for (int i = 0; i < t.Length; i++)
        var tPm = new Point3d();
        tPm.Interpolate(line.From, line.To, t[i]);
        tPm = new Point3d(Math.Round(tPm.X, 4), Math.Round(tPm.Y, 4),
            Math.Round(tPm.Z, 4));
        tempP.Add(tPm);
```

Listing 5.3: Pseudocode for interpolation of sub-element base points

5.2 Support Components

5.2.1 Boundary Conditions

The boundary condition component *BDCComponent* has seen some major improvements in terms of simplicity and ease of use. The previous boundary condition input has been replaced with buttons to lock different directions or rotation. The ”X”, ”Y” and ”Z” button will lock the respective direction they indicate and the same principle extends to the rotational button below. The component is shown in Fig. 5.4.

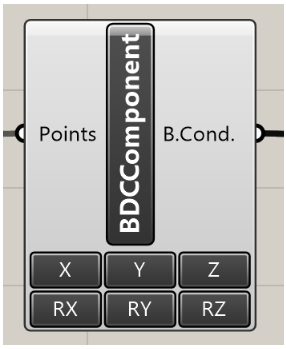
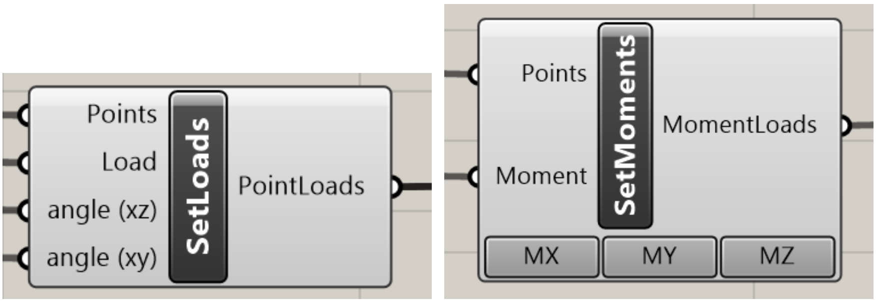


Figure 5.4: The boundary condition component for beams

As before it also takes in the point for which to apply the boundary conditions, and gives out the boundary conditions as a list of strings, but this time with the rotations added in the same manner as translational dofs.

5.2.2 Loads and Moments

The *SetLoads* component has seen little change from Truss and is described in Ch. 4.2.2, however the *SetMoments* component has been added. The two component can be seen in Fig. 5.5 below.



(a) SetLoads component.

(b) SetMoments component.

Figure 5.5: Support components for point load and moment load.

The *SetMoments* component requires the points to which the moment shall apply, and the magnitude of the moment load, given in Newtonmeters. The component allows for moments to be set about the X, Y and Z-axis of each node. Boolean toggles like on Fig. 5.5b decide whether the moments should be added. Moment magnitude can vary for each point, and if the input list of moment magnitudes is shorter than the input list for points, the last entry in the moment magnitude list will be used for all remaining points.

5.2.3 Deformed Geometry

The component *DeformedGeometry* receives the calculated deformations from the main component and redraws the geometry accordingly. The deformation can be scaled for illustrative purposes. The new nodal coordinates are found by using the base nodes and adding the calculated displacements (of u_x, u_y and u_z). At the end of each global for-loop (one for each element), the list of sub element nodes are used to create a polycurve, as can be seen in Lst. 5.4. The polycurve does not go "through" all nodes, but testing shows that the shape is very close to correct when using 4 or more elements.

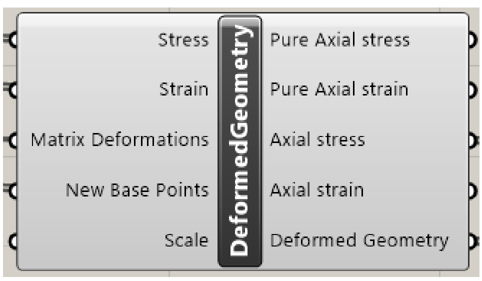


Figure 5.6: Deformed Geometry component for 3D Beam.

```
def = scale * def; //Calculate new nodal points
for (int i = 0; i < def.RowCount; i++)
    List<Point3d> tempNew = new List<Point3d>();
    for (int j = 0; j < n; j++)
        var tP = oldXYZ[i * n + j]; //original xyz

        //add deformations
        tP.X = tP.X + def[i, j * 6];
        tP.Y = tP.Y + def[i, j * 6 + 1];
        tP.Z = tP.Z + def[i, j * 6 + 2];

        tempNew.Add(tP); //replace previous xyz with displaced xyz
    //Create Curve based on new nodal points(degree = 3)
    Curve nc = Curve.CreateInterpolatedCurve(tempNew, 3);
    defCurve.Add(nc);
```

Listing 5.4: Generation of deformed geometry in 3D Beam

The component also receives the nodal strains and stresses from the calculation component in order find values per element. This is only relevant if the user would like axial stress or strain colored like on Fig. 5.7 or Fig. 5.8. This feature is still experimental and not fully tested as of yet, but generally shows reasonable results. The element value is found by first separating every third value in the list of stress/strain so that only pure axial stress/strain can be found. This new list then averages every two nodal values and skips over one entry whenever the correct number of sub-elements in one element has been calculated. These lists are outputted as ”Pure axial stress/strain”. The process is repeated for bending stress/strain about y and z axes. Afterwards, maximum stress/strain are calculated by Eq. 2.5.55. In time, this function might benefit from being moved to the calculation component instead.

```
for (int i = 0, ct = 0; s_avg.Count < el*n; i++)
    if (ct == n)
        ct = 0;
        continue;
    s_avg.Add((s[i] + s[i + 1]) / 2);
    ct++;
```

Listing 5.5: Averaging of strains/stresses in sub-elements

```
for (int i = 0; i < s_avg_x.Count; i++)
    if (s_avg_x[i] > 0)
        ss.Add(s_avg_x[i] + Math.Abs(s_avg_y[i]) + Math.Abs(s_avg_z[i]));
    else
        ss.Add(s_avg_x[i] - Math.Abs(s_avg_y[i]) - Math.Abs(s_avg_z[i]));
```

Listing 5.6: Maximum strains/stresses

Figure 5.7 and 5.8 shows colorized axial stresses for five top nodes vertically loaded. Red is compression, blue is tension and green is somewhere in the middle.

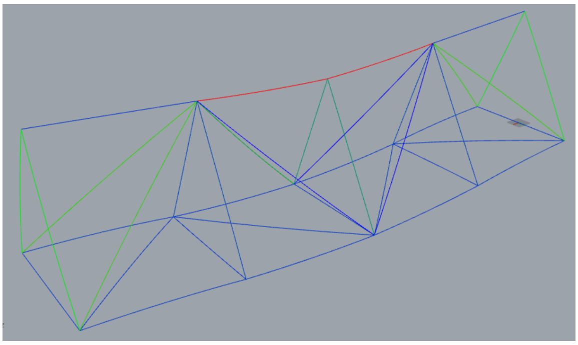


Figure 5.7: Coloring of pure axial stresses

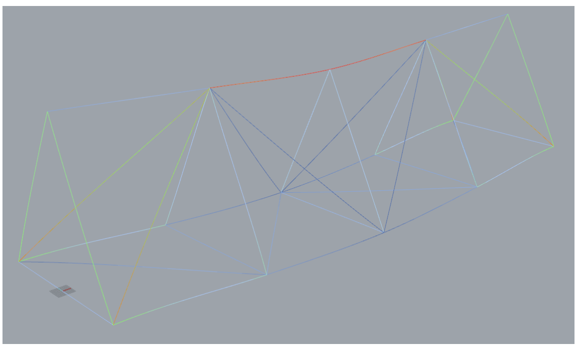


Figure 5.8: Coloring of maximum axial stresses

5.3 Analysis

5.3.1 Performance

To ensure the software package is a seamless addition to the parametric environment, calculations should be completed as quickly as possible. Grasshopper has the function to map the time-usage of each component in relation to other components. Unsurprisingly, the calculation component is the largest time drain out of all components. Its completion time typically varies between 50-99% of the main program, depending on structure complexity. This is also true for truss and shell software.

The runtime completion analysis on Fig. 5.9 revealed a bottleneck in the component. The Cholesky algorithm had the highest runtime of all code sections, by far, which was anticipated since the algorithm entails a fair number of calculations. In second place comes reduction of the global stiffness matrix and load list. This was more surprising, since construction of the global stiffness matrix is a considerably more extensive task. The reduce method shown used in Fig. 5.9 is the new reduction algorithm as shown in Lst. 5.1. It builds a new matrix of preallocated size and fills it, thus amending the misstep (in terms of runtime) from Lst. 4.5 in 3D truss.

Notice that the number of sub-elements in the figure is locked at four since this has typically given a decent representation of element deformation. Complex structures may need a slightly higher number. The affect from increasing the number of sub-elements are visualized in Fig. 5.13.

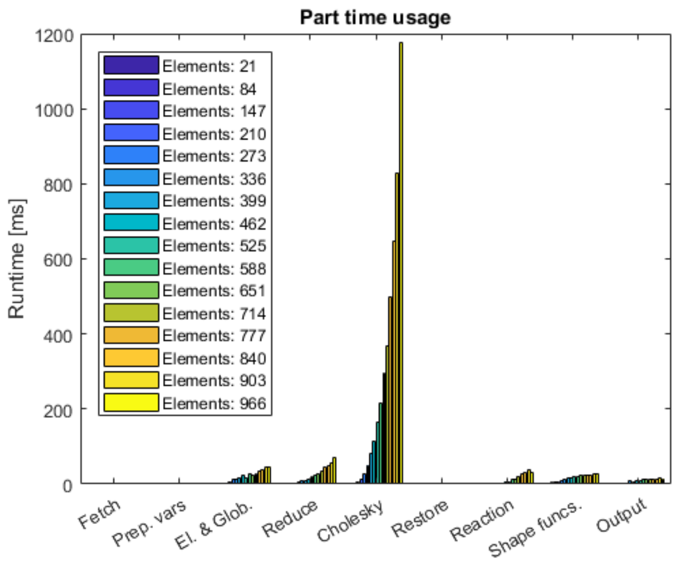


Figure 5.9: 3D Beam Calculation code section running time for 4 sub-elements

Table 5.1: Section partitioning in Fig. 5.9

Name	Description
Fetch	Fetch inputs from Grasshopper
Prep. vars	Preparation of variables for geometry, boundary conditions and loads
El. & Glob.	Construction of global stiffness matrix $\mathbf{K}^{\mathbf{G}}$
Reduce	Reduction of global stiffness matrix to $\mathbf{K}^{\mathbf{G}}_{\mathbf{r}}$
Cholesky	Cholesky Decomposition and substitutions
Restore	Restoration of deformation list
Reaction	Calculation of reaction forces
Shape funcs.	Calculation of internal displacements & strains
Output	Formatting of output

Math.NET supports both sparse and dense storage format. The main difference between the two is that sparse matrices only store nonzero entries, while dense matrices store all entries. Sparse matrices can be beneficial in handling of large matrices with few nonzero entries (Bell, 2013). However, Math.NET has not optimized their solvers for sparse matrices, and only the regular Solve() function will consistently manage to solve a system of equations. Fig. 5.10 shows the completion runtime for dense and sparse matrices, plotted against the number of reduced degrees of freedom in the structure. According to their documentation, Math.NET has not optimized their solver for sparse matrices.

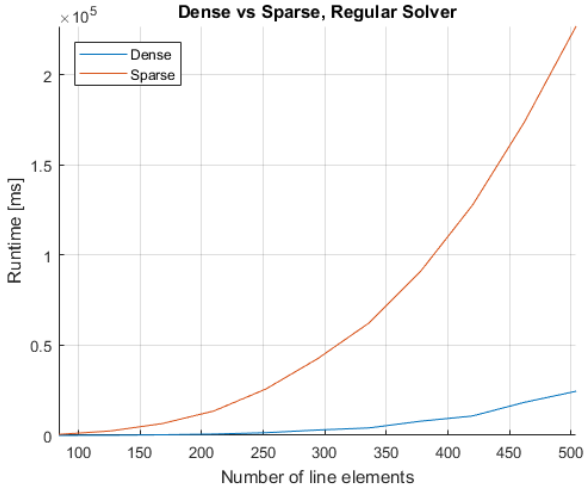


Figure 5.10: Runtime for dense vs sparse matrix.

A comparison was performed on five different solver algorithms provided by Math.NET. Among else it contains a Solve() method which is based on QR decomposition. Further documentation of the method could not be found. Next, it also has methods for Cholesky, QR, Svd and LU decomposition. These have all been tested for a varied number of rdofs, as shown on Fig. 5.11. It can be observed that Cholesky has a considerably smaller runtime than the other methods. This is examined further in the logplot on Fig. 5.12 which shows how the Cholesky algorithm performs in comparison to the others.

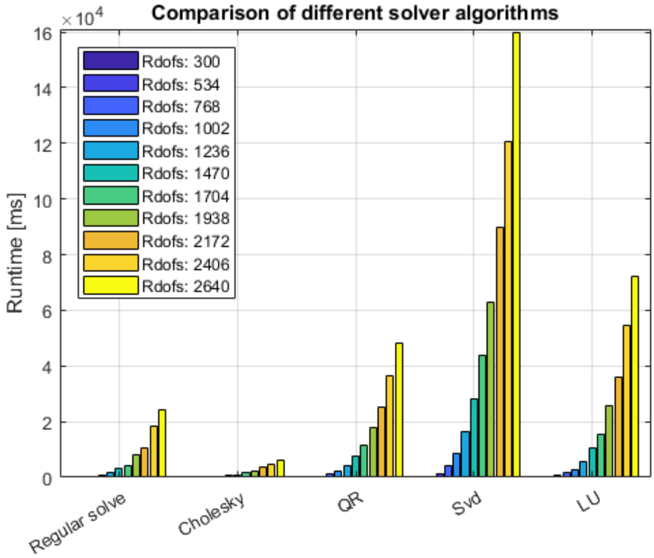


Figure 5.11: Runtime of various solver algorithms.

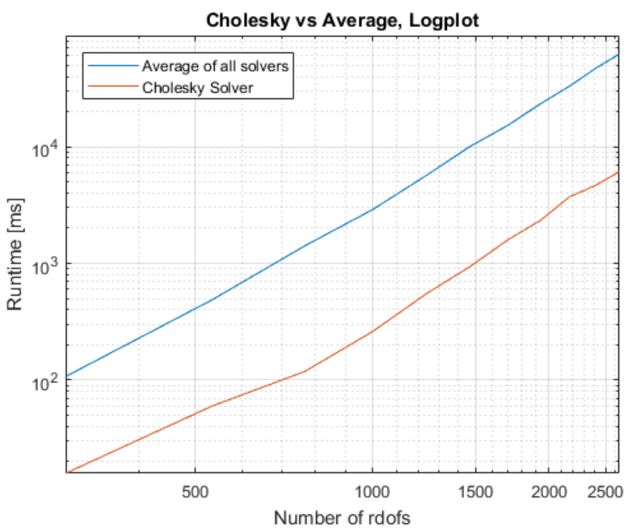


Figure 5.12: Logplot of cholesky vs other solvers.

The section containing shape functions includes both displacement and strain calculation of each sub-node in the structure. The displacement fields **N**, **dN** and **ddN** are found and postmultiplied by the relevant displacement vector **u**, as explained in Ch. 5.1.2-5.1.3. A test performed for different amounts of sub-elements can be seen in Fig. 5.13. The algorithm seems to have a running time close to $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$.

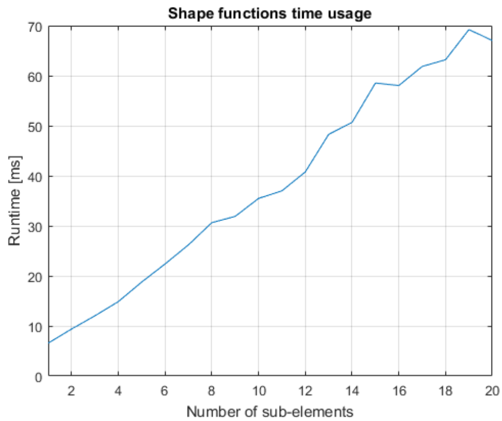


Figure 5.13: Plot of runtime for shape functions. 441 elements

As explained in Ch. 4.1.2, the 2D truss contained a self-made algorithm for Cholesky Decomposition and substitution. Lst. 4.6-4.8 was ported over to 3D Beam and compared against Math.NET's solution. As visualized on Fig. 5.14, the Math.NET solution is clearly better optimized, and likely utilizes multi-threading. This decrease in computation time led to 3D Beam adopting the Math.NET algorithm instead.

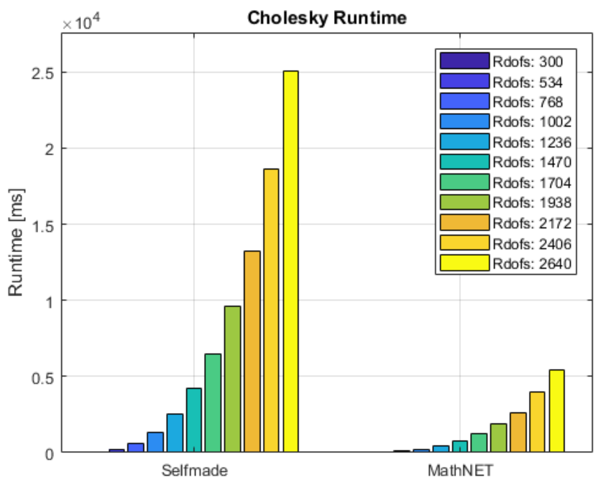


Figure 5.14: Comparison of Cholesky algorithms.

5.3.2 Accuracy

In this chapter, the beam software bundle is compared against Robot Structural Analysis, as well as an analytical solution where easily obtainable. All tests have been performed on beams of type HEB100 which has the material properties listed to the right. Note that Robot is based on Timoshenko beam theory while 3D Beam is based on Euler-Bernoulli. This may lead to small discrepancies between results.

E	210 000	[MPa]
Area	2 600	[mm^2]
Iy	$4.50 \cdot 10^6$	[mm^4]
Iz	$1.67 \cdot 10^6$	[mm^4]
G	80800	[MPa]
v	0.3	

Structure 1, load case 1

The first structure is a single horizontal beam of 10 meters, see Fig. 5.15. It is loaded with a vertical force of 10kN on the rightmost node. The left node is fully fixed while the right node is free. The analytical solution displacement w and rotation θ for such a beam are derived from the Euler-Bernoulli equation in Eq. 5.3.1 by integrating and applying boundary conditions.

$$EI\frac{d^4w}{dx^4} = q(x) \qquad \text{(Eq. 5.3.1)}$$

$$w(x) = -\frac{Px^2}{6EI}(3L - x) \qquad w_{max} = w(L) = -\frac{PL^3}{3EI} \qquad \text{(Eq. 5.3.2)}$$

$$\theta(x) = -\frac{Px}{6EI}(3L^2 - 3Lx + x^2) \qquad \theta_{max} = \theta(L) = -\frac{PL^3}{6EI} \qquad \text{(Eq. 5.3.3)}$$

Table 5.2: Displacements in right node for vertically loaded fixed beam

Solution	Displacements					
	u_x [mm]	u_y [mm]	u_z [mm]	θ_x [rad]	θ_y [rad]	θ_z [rad]
Analytical	0	0	-3527.337	0	0.529	0
Robot	0	0	-3531.260	0	0.530	0
Difference	0	0	-3.923 (0.1%)	0	0.01 (1.9%)	0
Beam 3D	0	0	-3527.337	0	0.529	0
Difference	0	0	0	0	0	0

The shape functions are used to calculate displacements within the element. As can be observed on Tab. 5.3, the displacements found by the displacement fields are identical to the ones found by the analytical formulas, Eq. 5.3.2-5.3.3. Tab. 5.3 checks the displacements at 1/4, 2/4 and 3/4 along element, from left to right.

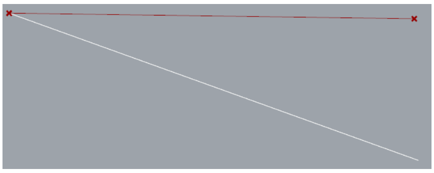
Table 5.3: Displacements within element for vertically loaded fixed beam

Solution	**Displacements** [mm] [rad]					
	x = 2500mm		x = 5000mm		x = 7500mm	
	u_z	θ_y u_z	θ_y u_z	θ_y		
Analytical	-303.13	-0.5621	-1102.29	-0.6614	-2232.1429	-0.2976
Beam 3D	-303.13	-0.5621	-1102.29	-0.6614	-2232.1429	-0.2976

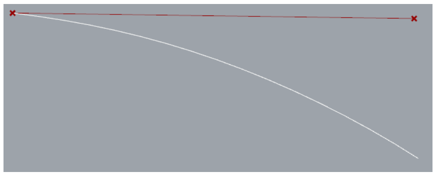
Table 5.4: Internal strain and stress at rightmost node

Solution	**Strain**	**Stress** [MPa]
Analytical	-0.005291	$-1111.\bar{1}$
Robot	N/A	-1112.35
Difference	N/A	1.24 (0.1%)
Beam 3D	-0.005291	$-1111.\bar{1}$
Difference	0	0

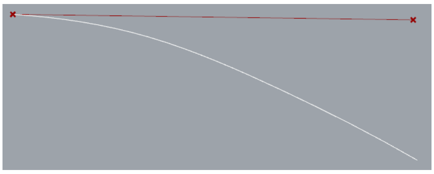
An important feature of the software is simulation of deformations. To this end, the element from load case 1 is shown with a deformation scale of 1 on Fig. 5.15. The figures show how the element gradually becomes more exact by incrementing the number of sub-divisions. This affects the displacement within each element, as explained in Ch. 5.1.2. Fig. 5.15f shows how Robot Structural Analysis displays the deformation.



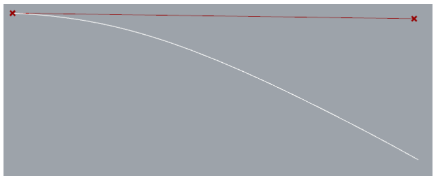
(a) 1 element, 1 sub-elements, 3D Beam



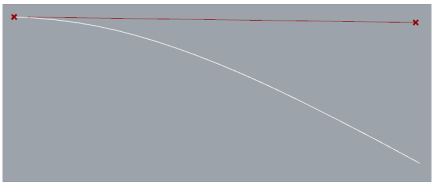
(b) 1 element, 2 sub-elements, 3D Beam



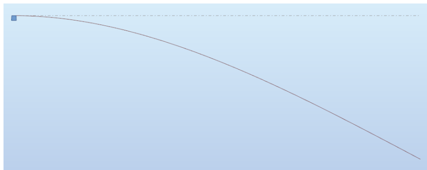
(c) 1 element, 3 sub-elements, 3D Beam



(d) 1 element, 4 sub-elements, 3D Beam



(e) 1 element, 5 sub-elements, 3D Beam



(f) Robot Structural Analysis

Figure 5.15: Element deformation for increasing number of sub-elements

Structure 1, load case 2

The structure and boundary conditions are similar to case 1. Instead of a point load, the structure is subjected to a uniformly distributed vertical load of 1 kN/m. Beam 3D can simulate uniformly distributed load cases by setting multiple point loads along the element, thereby splitting the element into multiple elements. The end node is loaded half as much as the other nodes since it only represents half the area. The situation is shown in Fig. 5.16 from Robot.

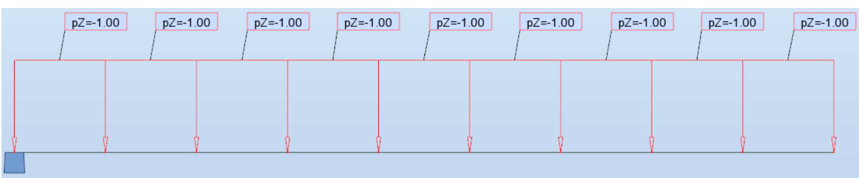


Figure 5.16: Uniformly distributed load

The relevant analytical equation is

$$w(x) = -\frac{qx^2}{24EI}(x^2 + 6L^2 - 4Lx) \qquad w_{max} = w(L) = -\frac{PL^4}{8EI} \qquad \text{(Eq. 5.3.4)}$$

As can be seen on Fig. 5.17, the beam software requires a vast amount of elements in order to properly converge towards the analytical solution.

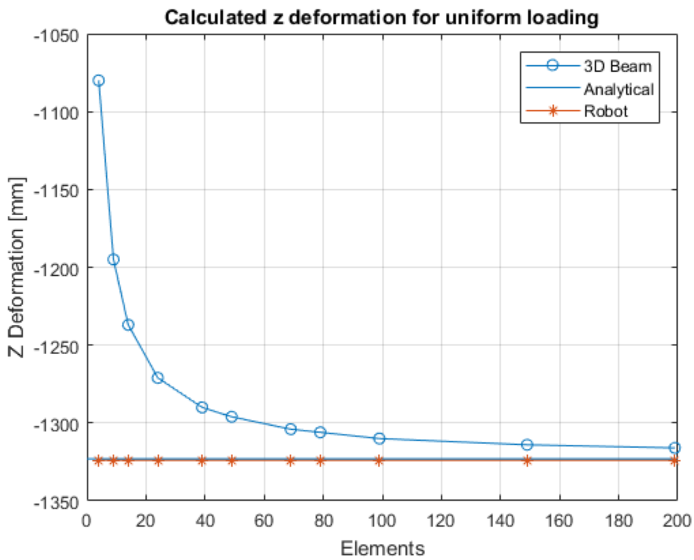


Figure 5.17: Deformation comparison of uniformly distributed load

Structure 2

The second structure is a span of 4 meters between two fixed endpoints. It is loaded with a vertical force of 10kN on the midpoint of the span (at 2 meters from left node). See Fig. 5.18.

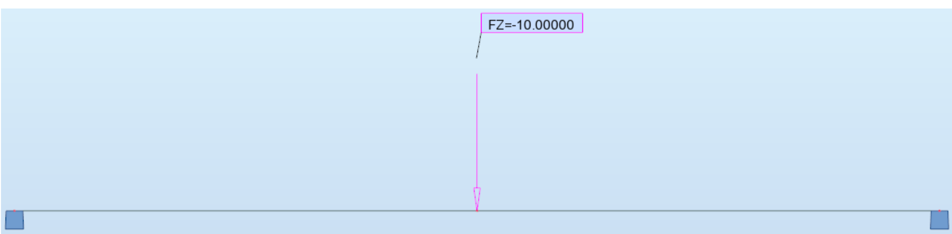


Figure 5.18: Span with vertical loading at midpoint

Table 5.5: Displacement, stress and strain in middle of span for vertical load

Solution	u_z [mm]	**Strain**	**Stress** [MPa]
Analytical	-3,52734	-0.000265	$-55.\bar{5}$
Robot	-3.53126	N/A	-55.617
Difference	-0.00392 (0.1%)	N/A	-0.062 (0.1%)
Beam 3D	-3.52734	-0.000265	$-55.\bar{5}$
Difference	0	0	0

Structure 3

The third structure is triangular structure spanning 4 meters between four fixed endpoints. All horizontal beams are 1 meter long. Distance from bottom to top of structure is 1 meter. Structure is loaded with a vertical force of 10kN on all top nodes. See Fig. 5.19.

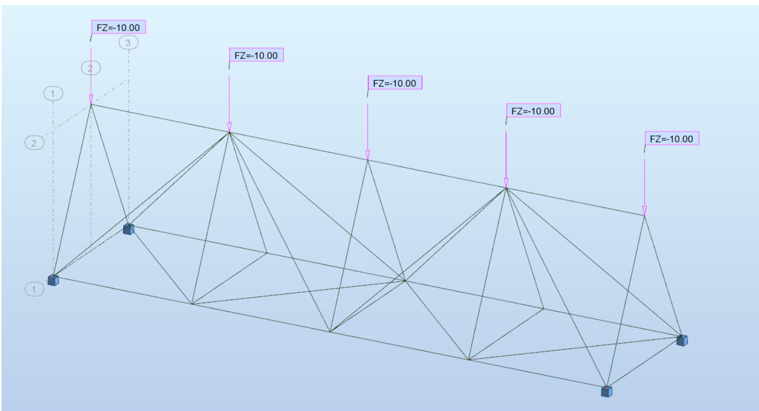


Figure 5.19: Complex beam structure loaded at top nodes

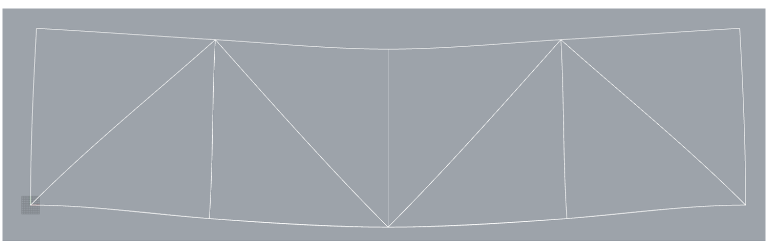


Figure 5.20: Deformed 2D view. Sub-elements set to 4, deformation scale set to 10

Table 5.6: Maximum displacement and stress for complex structure

Solution	$\mathbf{u_{max,z}}$ [mm]	$\mathbf{Stress_{max,x}}$ [MPa]
Robot	-0.1368	-7.12
Beam 3D	-0.1292	-7.04

These values remain the same even if the structure is divided into more elements. They were also found at the same nodes (top middle node for displacement and top node 1/4's and 3/4's for stress).

5.4 Discussion

Time usage of the calculation component is unsurprisingly bottlenecked by the Cholesky algorithm. As can be observed on Fig. 5.11, Cholesky is significantly faster than QR, Svd and LU, and is generally regarded as an able solver for Finite Element Analysis. The bar plot on Fig. 5.14 shows that Math.NET's solution is superior to our self-made algorithm, and plays a major role as to why the software bundles utilizes Math.NET.

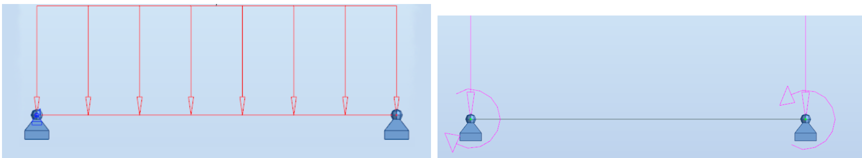
While the algorithms plotted in Fig. 5.11 are based on dense matrices, there would be advantages to employing sparse matrices instead. The global stiffness matrix will be very large for sizable structures, leading to a potential shortage of memory when solving the system of equations. Since sparse matrices only stores non-zero values, a lot of memory can be freed. Ch. 7 also briefly discuss solver algorithms.

As evident from the tests in Ch. 5.3.1, the 3D Beam software results are (usually) identical to the analytical solutions based on Euler-Bernoulli beam theory. This is not surprising, since the shape functions derived in Ch. 2.5.1 are the exact solutions of the Euler-Bernoulli beam equation. These shape functions are then used to derive the element stiffness matrix, as explained in Ch. 2.5.3.

It can be observed on Fig. 5.13 that the visualization in 3D Beam is very similar to Robot's when using 4 or more sub-elements. Based on this, the number of sub-elements can safely be set to 4 as default, with option to change as desired.

Although accurate for point loads and moment loads, Ch. 5.3.2 shows that the software is ill-equipped for handling of uniformly distributed loads. One way of solving this is to implement superposition of virtual moments (Barber, 2011). By this method, the system of equations would be solved for displacements as usual, then a correcting term would be added to those nodes subjected to uniformly distributed loading. This second term is the deflection resulting from adding virtual moments around these nodes. The moment magnitude is derived from simulating a fixed-end situation of the element. For a uniformly distributed load q_0, the load is transformed into F_z and M_y, where F_z is applied to both nodes, while M_y is positive for left node and negative for right node.

$$Fz = -\frac{q_0 L}{2} \qquad My = \frac{q_0 L^2}{12} \tag{Eq. 5.4.1}$$



(a) Actual situation (b) Interpreted situation

Figure 5.21: Uniform loading by superposition

For the more complex structure shown in Ch. 5.3.2, the maximum displacement and stress were slightly divergent. Although the tests that include an analytical solution are identical for point loads, it is hard to say whether this extends to complex structures. Further analysis is needed, especially since the test results in Tab. 5.6 show that 3D Beam potentially is on the ”unsafe side”.

As can be seen on Fig. 5.13, the number of sub-elements affects the running speed at a low exponential rate which is within expectations. The test results show some divergence from a trend line, but this is likely a result of a small sample size (ca. 5 per number of elements) and the short time usage (max 70 ms). Small optimization could be made at the cost of code readability. but as Fig. 5.3.1 shows, the shape function section is quick and scales better than the Cholesky algorithm.

Currently, the software is built on Euler-Bernoulli rather than Timoshenko beam theory. Accounting for shear deformations might be more accurate, but would come at the expense of running time. Since target user of this software is architects rather than structures engineers, Euler-Bernoulli has been deemed to give sufficient accuracy. A consideration for further work would be adding dynamics, in which case implementing Timoshenko would have to be reassessed.

The strains and stresses are one-dimensional for much the same reasons as for applying Euler-Bernoulli. Since the target users are architects rather than structural engineers, the solution should be approximately correct and quick rather than exactly correct and slow.

5.5 Beam Summary

The beam software bundle is similar to truss in many aspects. It consists of five components, a main component for calculation and four support components for point loads, moments, boundary conditions and generating deformed geometry. However, the main component has a more sophisticated transformation matrix, more dofs per node, and applies shape functions for calculations within the element. Furthermore, the user friendliness of the components has been vastly improved by adding toggles to the graphical layout of the *Boundary Conditions* component and the *SetMoments* component.

Displacements, strains and stresses are very accurate for point loads and moments, but would benefit from a proper implementation for uniformed loading. The deformed geometry component incorporates preparation coloring of axial strains and stresses, but is still dependent on a small cluster of Grasshopper components in order to give color to the geometry. Furthermore, internal forces are not calculated and would be very relevant for future work.

For structures of less than 1000 elements, the calculation component will not take longer than approximately 1.2 seconds. The main bottleneck is the Cholesky algorithm for solving of displacements, and cannot be easily optimized without extensive refactoring. This would be a goal for further work.

The component calculates one-dimensional stresses and strains, and is based on the Euler-Bernoulli beam theory. The software can only analyze line elements, meaning that curved beams are not supported. Having 6 dofs per node has been determined as necessary so as to account for general load applications, even though this leads to slightly slower calculations.

Chapter 6

Shell Calculation Software

The shell calculation software consists of four components, where the three support components are the Boundary Conditions (*Shell BDC*), Point Loads (*PointLoads Shell*) and Deformed Shell (*DeformedShell*). These provides the Shell Calculation (*ShellCalculation*) component with the necessary inputs and presents a deformed preview of the deformed structure. The full source code for Shell can be found in Appendix D. Their relation can be simplified as

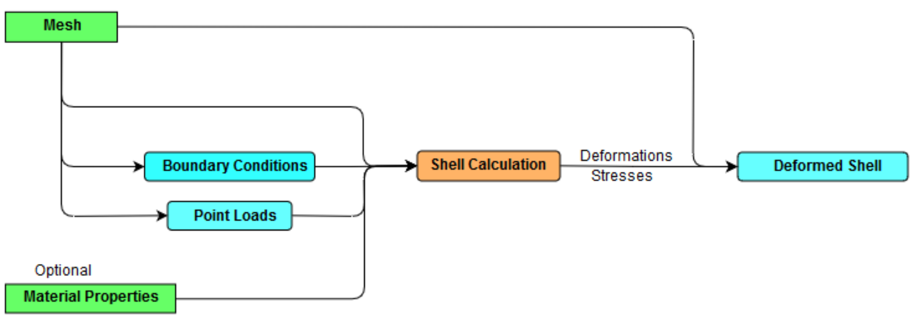


Figure 6.1: Overall organization for the shell calculation software

The shell software employs the triangular CST-Morley element as described in Ch. 2.6.4 - 2.6.6. This element has three translational deformations in each node, and rotation about each of the three edges, which results in 12 dofs per element.

6.1 Calculation Component

The main part of the shell software is the *ShellCalculation* component shown in Fig. 6.2.

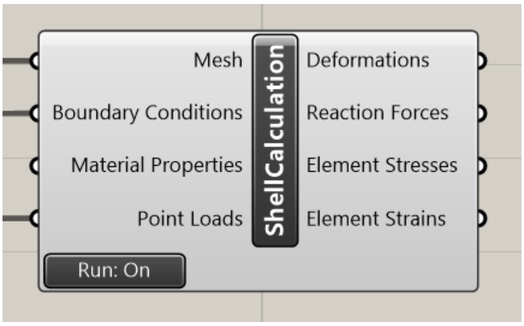


Figure 6.2: The main component for shell calculation

The inputs to this component are:

Mesh - The meshed structure must be a triangular mesh for the calculation to compute correctly, which means the elements needs to be triangles. This is generated in Grasshopper.

Boundary conditions - These are given by the *Shell BDC* component, which will be explored in Ch. 6.2.1.

Material Properties - The material properties themselves and their usage are more thoroughly covered in Ch. 2. They are given as a string of numbers in the following order:

1. Young's Modulus, also called the Elastic Modulus, denoted E.
2. Poisson's ratio ν.
3. The shell thickness t.
4. Shear modulus, or modulus of rigidity. If this is not given in a string, the program will automatically set it as $\frac{E}{2(1+\nu)}$.

Note that all material properties are assumed constant in this software. In the case where nothing is given as input to Material Properties, the string is preset assuming steel (E=200000 MPa and ν=0.3) with 10mm thickness. The preset string is of the format "200000,0.3,10".

Point Loads - Point loads are given by the support component *Point Loads*. This component will be explained in Ch. 6.2.2.

The outputs from *ShellCalculation* are:

Deformations - The global deformations for all dofs is formatted as a list with x, y and z translation for each node, followed by all the rotations. This means that all translations are ordered as they are found from the list of vertices in the given mesh. After all translations are listed, the rotational deformations are listed in the order they are found from the list of faces. If one face has three vertices A, B and C, the edges are ordered as edge AB, followed by edge BC, followed by edge CA. None of the dofs (translational or rotational) occurs more than once in the list. Also, the constrained dofs are included and will naturally have a value of zero.

Reactions - The reaction forces are calculated from the global deformations, and therefore follow the same ordering as the deformations. Reactions forces are given as point forces in respectively x, y and z direction for all nodes, followed by all moment forces in the edges. This means that the list of reaction forces will be the same length as the deformation list. Some of the more important forces in this list will be the ones that corresponds to the zeroes in the deformation output, as these are the forces in the supports. Note that the list of reaction forces also includes the applied loads (the action forces). In this way one can easily retrieve the needed forces as the reaction forces corresponds to the zeroes in the deformation lists, and the applied forces generally corresponds to deformations larger than zero.

Element Stresses - The stresses are given per element and in local axes. The reason for this is explained in Ch. 6.1.1. Stresses are arranged according to faces since each face represents an element, and therefore in the same order as the face-list from the given mesh. Each face has a total of six stresses, ordered as follows

$$\begin{bmatrix} \sigma_x^m & \sigma_y^m & \tau_{xy}^m & \sigma_x^b & \sigma_y^b & \tau_{xy}^b \end{bmatrix}$$

The letter m denotes membrane and b bending.

Element Strains - The strains are given in the same order as the stresses and also in local axes. Since the stresses and strains are a linear combination of another, as seen from Eq. 2.6.20 and 2.6.45, a relation can easily be spotted for the two lists. The strains are ordered as

$$\begin{bmatrix} \varepsilon_x^m & \varepsilon_y^m & \gamma_{xy}^m & \varepsilon_x^b & \varepsilon_y^b & \gamma_{xy}^b \end{bmatrix}$$

The tasks handled in the main calculation component can be separated into three groups, namely pre-processing, processing and post-processing. The pre-processing will include the preparations done before and outside the main calculation component, e.g. the preparation of boundary condition and loads. In the same manner, most of the preparation of the results after and outside the main component is part of the post-processing.

The calculation component can be considered to work in nine steps. Four of these steps are parts of the pre-processing, one is the processing, and the remaining four belongs to the post-processing. This is visualized in Fig. 6.3. It can also be observed that post- and pre-processing are not necessarily separated in terms of dependencies.

6.1.1 Pre-Processing

Fetch Input

The mesh data structure in grasshopper gives easy access to faces and vertices (Ramsden, 2014), however it does not store the edges of the faces. As the CST-Morley element shown in Fig. 2.14 has rotational dofs around each edge, the edges needs to be retrieved. The way this is done is shown in the pseudo code in Lst. 6.1.

```
// Number of edges from Euler's formula
No. of Edges = No. of nodes + No. of faces - 1
edges = create list with No. of edges entries

foreach face in faces
    Create all possible lines //(eg. AB and BA)
    if (the edgelist does not already contain the edge)
        add edge to edges list

    repeat for all edges
```

Listing 6.1: Pseudocode for extracting the edges of each element

To make sure there are no duplicate nodes a very similar procedure is utilized to create a list of unique nodes.

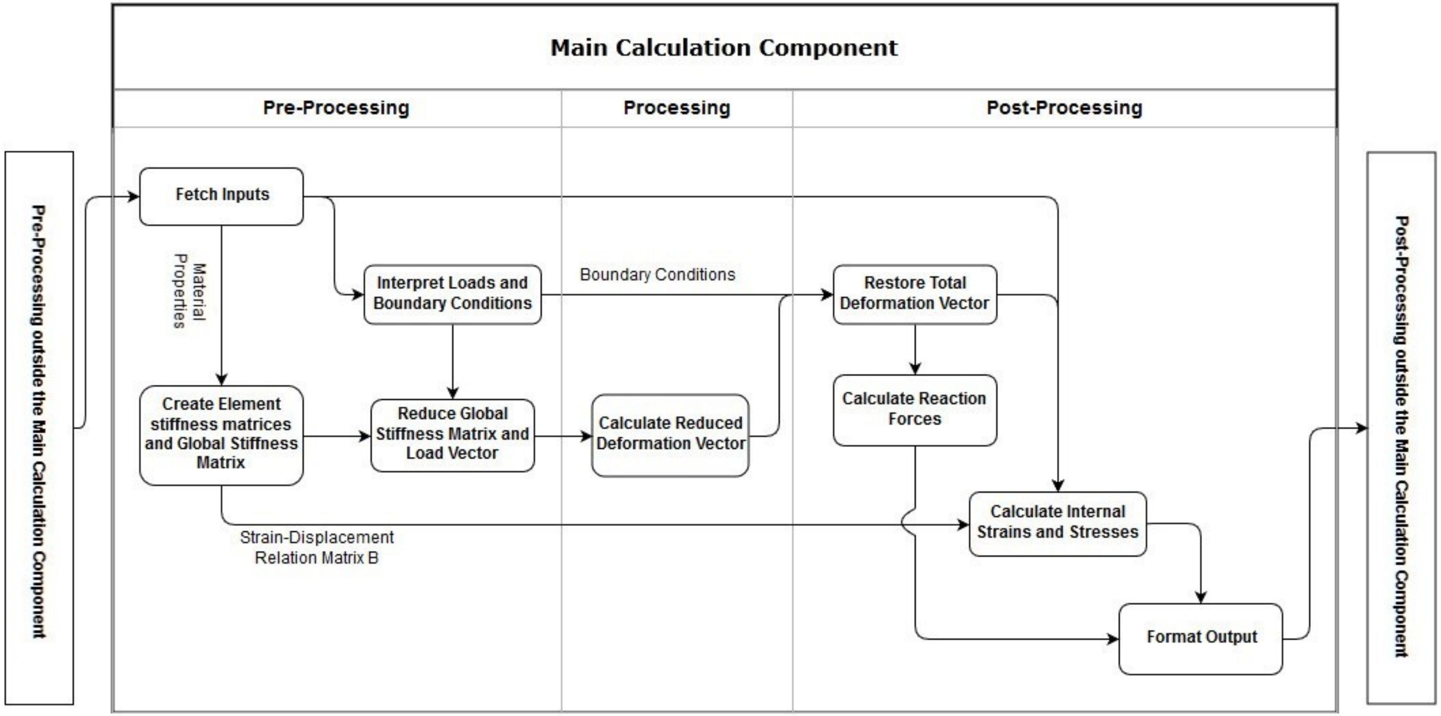


Figure 6.3: Simplified workflow of the main component

Interpret Loads and Boundary Conditions

The next step is to interpret the boundary conditions and applied point loads, this is done by the methods *CreateLoadList* and *CreateBDCList*. The creation of the load list is relatively straight forward. It is simply a matter of decomposing the string given by the *PointLoad* component, described in Ch. 6.2.2, converting them into numbers and place them in a load list according to the dofs ordering. The dofs order is given by the unique node list followed by the list of edges.

The boundary conditions is given as a list of strings, described in Ch. 6.2.1. The given strings are handled as described in the pseudo code in Lst. 6.2 below. Note that if there are fixed edges, they are given as edge indices and gathered in one string at the end of the list input.

```
// Initiating the bdc_value with 1's, where 1 = free and 0 = clamped
bdc_value = list with (No. of uniquenodes * 3 + No. of edges) entries,
    filled with 1's

foreach BDCstring input
    if BDCstring does not contain ":" // indicating this is fixed edges
        store the edge indices
    else
        store the clamped directions
        store the specified point

// set stored clamped directions' corresponding value in bdc_value to 0
foreach stored point
    set the bdc_value to 0 for each of the clamped directions
        corresponding to point placement in uniquenodes

// set the stored edge indices' corresponding values in bdc_value to 0
foreach stored edge
    set the corresponding rotational dof in bdc_value to 0

// bdc_value will have a 0 for every clamped dof, and 1's otherwise
```

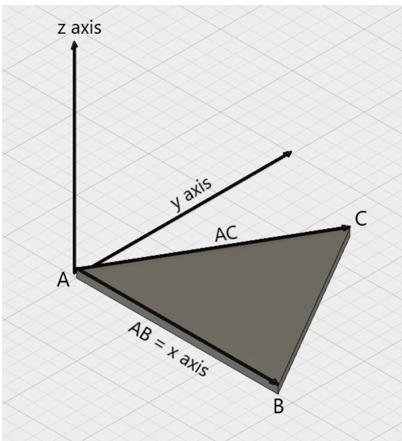
Listing 6.2: Pseudocode for creating the boundary condition list

Create Element Stiffness Matrices and Global Stiffness Matrix

With the load and boundary condition lists established, the global stiffness matrix is next. To create the global stiffness matrix, each element stiffness matrix has to be derived. The CST-Morley element can be assembled as shown in Eq. 2.6.105, for which the membrane (CST) and the bending (Morley) stiffness matrices has to be found. Both matrices are dependent upon the coordinates of each of the three nodes of the element, as can be seen from Eq. 2.6.69 and Eq. 2.6.102, where among else the area is needed.

Because of the dependency on coordinates, creating the element stiffness has to be repeated for every element. The process of establishing this has therefore been delegated to its own method called *ElementStiffnessMatrix*. The first issue to overcome is that the coordinates at hand is related to the global coordinate system. A transformation matrix is therefore necessary, and it will also be unique for each element. The method of direction cosines shown in Eq. 2.7.18 can be used. This is because the three points needed to define the local axes are given by the element as vertices.

The local axes can easily be defined by appointing the first edge AB as the local x axis, thereafter using the cross product to get the other axes as illustrated in Fig. 6.4, the procedure will be as follows.



By the cross product and the right hand rule, the axes becomes

x axis $= AB$

z axis $= AB \times AC$

y axis $=$ x axis $\times$ z axis

Figure 6.4: Defining local axes

Where A, B and C is defined as

$$A = (X_1, Y_1, Z_1) \qquad B = (X_2, Y_2, Z_2) \qquad C = (X_3, Y_3, Z_3)$$

The defining of the local axes is a straightforward matter to implement with code. The implementation used in Matlab can be examined in Appendix D.1, which outputs the full expressions for the direction cosines in the C# language. These expressions can now easily be evaluated when provided with the coordinates of the three nodes.

The transformation matrix can now be established from Eq. 2.7.18. The transformation from global (X, Y, Z) coordinates to local (x, y ,z) is given by

$$\mathbf{A}_l = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} cos(x,X) & cos(x,Y) & cos(x,Z) \\ cos(y,X) & cos(y,Y) & cos(y,Z) \\ cos(z,X) & cos(z,Y) & cos(z,Z) \end{bmatrix} \begin{bmatrix} X_1 \\ Y_1 \\ Z_1 \end{bmatrix} = \mathbf{t}\mathbf{A}_g \qquad \text{(Eq. 6.1.1)}$$

Where $\mathbf{t}$ is the transformation matrix corresponding to each element. There are several ways to use this to transform the coordinates from global to local, but the method chosen here is through assembling the global coordinate matrix as

$$\mathbf{v}_g^e = \begin{bmatrix} X_1 & X_2 & X_3 \\ Y_1 & Y_2 & Y_2 \\ Z_1 & Z_2 & Z_3 \end{bmatrix} \qquad \text{(Eq. 6.1.2)}$$

And transforming it into local coordinates as

$$\mathbf{v}_l^e = \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ z_1 & z_2 & z_3 \end{bmatrix} = \mathbf{t} \begin{bmatrix} X_1 & X_2 & X_3 \\ Y_1 & Y_2 & Y_2 \\ Z_1 & Z_2 & Z_3 \end{bmatrix} \qquad \text{(Eq. 6.1.3)}$$

The local coordinates for the element is now established, and the stiffness matrices can now be established.

First the task is to establish the Morley triangle stiffness matrix, which can be established from Eq. 2.6.102. The process of deriving the Morley triangle as shown in Ch. 2.6.5 is a tedious process to repeat for every element. Instead, the Matlab script in Appendix D.2 was made to create an explicit expression for the $\mathbf{B}_K$ matrix from Eq. 2.6.100 and export it as C# code. The equation reads

$$\mathbf{k}_b^e = \int_{A_e} \mathbf{B}_K^T \mathbf{D} \mathbf{B}_K dA$$

While the $\mathbf{D}$ matrix is constant, the $\mathbf{B}_K$ matrix requires the local x and y coordinates in addition to γ_m, μ_m and α_m from Eq. 2.6.90 to 2.6.92. These equations read

$$\gamma_m = \frac{c_m x_{32} - s_m y_{23}}{2A}$$

$$\mu_m = \frac{c_m x_{13} - s_m y_{31}}{2A}$$

$$\alpha_m = \gamma_m + \mu_m$$

The variables c_m and s_m can be seen from Fig. 2.13 and the notations thereunder. The variables γ_m, μ_m and α_m has been calculated unambiguously as illustrated in the pseudo code in Lst. 6.3, with inspiration from (Bell, 2013).

```
x13 = x1 - x3
x32 = x3 - x2
y23 = y2 - y3
y31 = y3 - y1
Area = Area of triangular element

foreach edge i of triangle (i = 1,2,3)
    length = length of edge i
    if (x_{i+1} > x_i) //note that x and y rotates cyclic -> x4 = x1
        c_m = (x_{i+1} - x_i)/ length
        s_m = (y_{i+1} - y_i)/ length
    else if (x_{i+1} < x_i)
        c_m = (x_i - x_{i+1})/ length
        s_m = (y_i - y_{i+1})/ length
    else
        c_m = 0
        s_m = 1

    γ_m = (c_m*x32 - s_m*y23)/(2*Area)
    μ_m = (c_m*x13 - s_m*y31)/(2*Area)
    α_m = γ_m + μ_m
```

Listing 6.3: Pseudocode for creating the boundary condition list

All the variables required for calculating $\mathbf{B}_K$ has now been determined and can now be used to calculate the stiffness matrix as

$$\mathbf{k}_b^e = \mathbf{B}_K^T \mathbf{D} \mathbf{B}_K A_e \qquad \text{(Eq. 6.1.4)}$$

Where with regard to Eq. 2.6.85

$$\mathbf{D} = \frac{Eh^3}{12(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} = \frac{h^3}{12}\mathbf{C} \qquad \text{(Eq. 6.1.5)}$$

The membrane part of the shell element is represented by the CST triangle. Which also receive its explicit expression for $\mathbf{B}_m$ matrix from the Matlab script in Appendix D.2. The $\mathbf{B}_m$ matrix is only dependent on the elements coordinates and is therefore calculated immediately after the $\mathbf{B}_m$ matrix is defined. It should also be noted that both $\mathbf{B}$ matrices are saved for each element, this to calculate the strain easily in the post-processing.

The stiffness matrices for membrane and bending are now assembled as shown in Eq. 2.6.105, and rearranged to correspond to the following deformation order

$$\mathbf{v}^e_{shell} = \begin{bmatrix} u_1 & v_1 & w_1 & \phi_4 & u_2 & v_2 & w_2 & \phi_5 & u_3 & v_3 & w_3 & \phi_6 \end{bmatrix} \qquad \text{(Eq. 6.1.6)}$$

The element stiffness matrix will thus look like

$$\mathbf{k}^e_{local} = \begin{bmatrix} \mathbf{k}_{11} & \mathbf{k}_{12} & \mathbf{k}_{13} \\ \mathbf{k}_{21} & \mathbf{k}_{22} & \mathbf{k}_{23} \\ \mathbf{k}_{31} & \mathbf{k}_{32} & \mathbf{k}_{33} \end{bmatrix} \qquad \text{(Eq. 6.1.7)}$$

Where $\mathbf{k}_{ij}$ is the stiffness relation between node/edge i and j.

The last step for the element stiffness matrix is to transform it back to global coordinates, so as it can be assembled into the global stiffness matrix. This is done by diagonally stacking the transformation matrix from Eq. 6.1.1 to fit the corresponding deformation order. As the rotations is about the edges, the rotation will be the same as long as the translational dofs are transformed correctly, and hence does not need to be transformed. The transformation matrix therefore assembled as

$$\mathbf{T} = \begin{bmatrix} \mathbf{t} & 0 & \mathbf{0} & 0 & \mathbf{0} & 0 \\ \mathbf{0} & 1 & \mathbf{0} & 0 & \mathbf{0} & 0 \\ \mathbf{0} & 0 & \mathbf{t} & 0 & \mathbf{0} & 0 \\ \mathbf{0} & 0 & \mathbf{0} & 1 & \mathbf{0} & 0 \\ \mathbf{0} & 0 & \mathbf{0} & 0 & \mathbf{t} & 0 \\ \mathbf{0} & 0 & \mathbf{0} & 0 & \mathbf{0} & 1 \end{bmatrix} \qquad \text{where } \mathbf{t} \text{ and } \mathbf{0} \text{ are 3x3 matrices} \qquad \text{(Eq. 6.1.8)}$$

Now the global element stiffness matrix can be calculated from Eq. 2.7.6, which reads

$$\mathbf{K^e_{global}} = \mathbf{T^T k^e_{local} T}$$

Which is straightforward to implement with code using Math.Net matrix multiplication.

The next major step is to assemble the element stiffness matrices into a global stiffness matrix. This operation is implemented in a similar fashion as Eq. 4.1.17, except the placement becomes more complex as the edges also has to be placed correctly. The stiffness matrices relating the nodal dofs are placed according to the element node indices in the unique node list. This ensures the correct stiffnesses are added together. In a similar way is the rotational stiffnesses placed according to edge indices in the edge list, and the stiffnesses relating nodes to rotation are placed based on both edge and nodal indices. The procedure is quite messy and may be hard to grasp as there are a lot of placement details that has to be correct. Nevertheless, the principle is the same as for both beam and truss, and can with some concentration be properly implemented.

Reduce the Global Stiffness Matrix and Load Vector

The last step in the pre-processing is the reducing of the global stiffness matrix and load list. The reducing requires the boundary conditions to check if the current row and column shall be removed. The method used utilizes two for loops, one for each row and one for each column. The current method was not always the utilized one, but was optimized due to excessive time usage, this is further examined in the analysis in Ch. 6.3. Both the reduced global stiffness matrix and the reduced load list is pre-allocated for time optimization. They are initialized with the sum of the bdc_value list, described in Lst. 6.2. This is done as all the free dofs have the value 1 and the clamped 0, the sum therefore gives the correct size of the reduced matrix and load list.

The method works by running through all the rows in the outer for loop, where the rows that correspond to the value 1 in the bdc_value list, is taken to the inner for loop. The rows that reach the inner for loop is looped through once more to check if the column corresponds to the value 1 in the bdc_value list. The rows taken to the inner loop is not necessarily looped entirely through, this is because the global stiffness matrix is known to be symmetric. By this reason only the lower triangular part of the matrix is looped through.

If the column in the inner loop corresponds to a bdc_value of 1, the current element in the global stiffness matrix is copied to the reduced global stiffness matrix. the value is inserted into both the lower triangular placement and the symmetric upper triangular placement. This method is implemented similar to the pseudo code shown in Lst. 6.4 below.

```
oldSize = length of load list
newSize = sum of bdc_value
K_red = create matrix with newSizexnewSize filled with 0
load_red = create list with newSize entries filled with 0
for (row = 1 to oldSize)
    skipR = 0
    if (bdc_value(row) == 1) //is the row corresponding to a free dof?
        for (col = 1 to row)
            skipC = 0
            if (bdc_value(col) == 1) //is the col corresp. to a free dof?
                K_red(row-skipR,col-skipC) = KG(row,col)
                K_red(row-skipC,col-skipC) = KG(row,col)
            else
                skipC += 1

    load_red(row-skipR) = load(row)
    else
        skipR += 1;
```

Listing 6.4: Pseudocode for creating the boundary condition list

6.1.2 Processing

Based on the results from Ch. 5.3.1 the Math.Net Cholesky solver was chosen to solve the global deformation-load relation. The solving of the global shell problem reads

```
Vector<double> def_reduced = K_red.Cholesky().Solve(load_red);
```

Listing 6.5: Solving the linear system of equations for shell structure

Which gives the reduced deformation list, where the word "reduced" indicates that the 0-value deformations corresponding to the clamped dofs has not been inserted yet. This may be the easiest line to implement in the shell code, however it is often the most time consuming by far, as shall be seen in Ch. 6.3.

6.1.3 Post-Processing

Restore Total Deformation Vector

The restoration of the complete deformation list is done by first creating a list of zeroes, with the length of the bdc_value list from Lst. 6.2, then looping through the bdc_value list and inserting the deformation from def_red for each value that is 1. Like this, the total deformation list is assembled with displacements at correct indices.

Calculate Reaction Forces

The calculation of the Reaction forces is also a straightforward process. As shown in Eq. 2.3.1, it is done by right-multiplication of a matrix and a vector. This is done as shown in Lst. 6.6 below

```
Vector<double> reactions = K_tot.Multiply(def_tot);
```

Listing 6.6: Solving for the Reaction forces

It should also be noted that since the total global stiffness matrix is used with the total deformation vector, the result will include the action forces, which in this case means the loads. This is done because it may be useful to get the applied loads together with the reaction forces for later inspection or use.

Calculate Internal Strains and Stresses

The strains and stresses for each element is local values, therefore the transformation matrix has to be established once again. This is done just as in Ch. 6.1.1 when the element stiffness matrix was established and will therefore not be repeated.

The next step is to use the stacked $\mathbf{B}$ matrices also from Ch. 6.1.1 when the element stiffness matrices was established. For each element the corresponding $\mathbf{B}_m$ and $\mathbf{B}_K$ matrices for respectively membrane and bending stresses are extracted. The corresponding displacements are also extracted from the total deformation list as $\mathbf{v}_m$ and $\mathbf{v}_b$, rearranged correctly and transformed to local deformations so that they can be combined with the $\mathbf{B}$ matrices.

The strains are calculated separately for the membrane and bending in accordance with Eq. 2.6.80, which gives the two equations

$$\boldsymbol{\varepsilon}_m = \begin{bmatrix} \varepsilon_{x,m} \\ \varepsilon_{y,m} \\ \gamma_{xy,m} \end{bmatrix} = \mathbf{B}_m \mathbf{v}_m \qquad \text{and} \qquad \boldsymbol{\varepsilon}_b = \begin{bmatrix} \varepsilon_{x,b} \\ \varepsilon_{y,b} \\ \gamma_{xy,b} \end{bmatrix} = -\frac{t}{2} \mathbf{B}_K \mathbf{v}_b \qquad \text{(Eq. 6.1.9)}$$

With the strains established, the stresses can be calculated from Eq. 2.6.30 and 2.6.45. The equations becomes

$$\boldsymbol{\sigma}_m = \begin{bmatrix} \sigma_{x,m} \\ \sigma_{y,m} \\ \tau_{xy,m} \end{bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_{x,m} \\ \varepsilon_{y,m} \\ \gamma_{xy,m} \end{bmatrix} \qquad \text{(Eq. 6.1.10)}$$

$$\boldsymbol{\sigma}_b = \begin{bmatrix} \sigma_{x,b} \\ \sigma_{y,b} \\ \tau_{xy,b} \end{bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_{x,b} \\ \varepsilon_{y,b} \\ \gamma_{xy,b} \end{bmatrix} \qquad \text{(Eq. 6.1.11)}$$

The strains and stresses are then placed into the internal strains and internal stresses list, and ordered according to the face list. In this way the output gives the membrane and the bending strains and stresses for each element in the same order as the faces are listed.

Format Output

The lists of total deformations, reaction forces, internal strains and internal stresses are simply given as outputs as they already are arranged as desired. If the ”Run”-button says ”Off”, all the outputs are set to zero.

6.2 Support components

The shell software includes three support components. For the pre-processing there is a component to defined boundary conditions namely *Shell BDC*, and one for defining point loads namely *SetLoads Shell*. The last support component is named *DeformedShell*, and will attempt to visualize the results.

6.2.1 Boundary Conditions

The *Shell BDC* component, seen in Fig. 6.5 takes two inputs, namely Points and Mesh. The points is the points that shall be fixed in one or more directions. The nodes for a CST-Morley element has no defined rotational dofs and therefore can only be clamped in translation. The edges of an element can however be fixed in rotation. Therefore, to clamp an edge in this software, at least two points must be given as input and they must have an element edge connecting them. If the ”Fix Rotation” button is activated, the component will attempt to find all the edges connecting the given points, and defining them as clamped in the output. To fix rotational dofs on the edges, the mesh structure is needed as an input for the component to be able to locate any edges.

The ”X”, ”Y” and ”Z” button on the component simply indicates which directions are set as clamped, in Fig. 6.5 an example can be seen where all dofs are set as clamped.

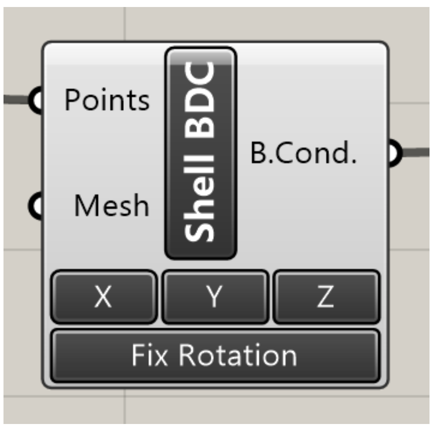


Figure 6.5: Shell BDC Component and example output

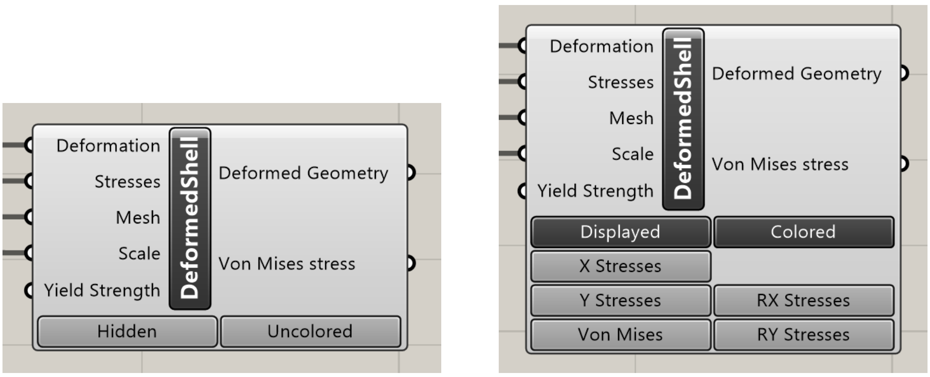
The output format is similar to that of the truss and beam software where the strings are formatted as x, y and z coordinate, followed by the corresponding condition values. The condition values can either be 1 for free or 0 for clamped. At the end of the list is one entry with the fixed edges (the fixed rotations).

6.2.2 Point Loads

The nodal loading component works in the exact same way as for the truss software and has been explained in Ch. 6.5, and will not be repeated here. Note that for distributed loads to be applied, it has to be transformed into points and nodal loads to be applied with this component.

6.2.3 Deformed Geometry

The support component *DeformedShell* in its hidden state is shown in Fig. 6.6a, and in its displayed and colored state in Fig. 6.6b.



(a) The deformed structure is hidden

(b) Deformation and stresses is visualized

Figure 6.6: The DeformedShell component in different states

There are only two required inputs, namely ”Deformation” and ”Mesh”, the rest are optional. The deformation input is the outputs given from the main shell calculation component, while the mesh is the same that is given as input to the main calculation component. The ”Scale” input is preset to 10 if no other input is given as a scaling parameter. The scaling work by multiplying, so for a deformation of e.g. 3 mm, and scale 100, the component will show 300 mm deformation.

For the ”Von Mises stress” output to supply any values, the stresses must be given as input. The stresses are the ”Element Stresses” from the main calculation component. The

"Von Mises stress" will give the Von Mises yield criterion for each element, and is ordered according to the face list in the mesh structure.

The coloring of the deformed structure requires stresses to run, and can take an optional yield strength. A maximum and a minimum value will be set inside the component, where the maximum defines red, and minimum defines blue. Other stresses will be interpolated and colored accordingly between these values. The yield strength can either be given as one positive number, which will be interpreted as the maximum positive yield strength and the minimum will be set to the equivalent negative number. Another option is to give a list with two different values, and the component will automatically set the minimum and maximum yield strength regardless of the order. If no value is given, the maximum

For the coloring of the mesh the component uses node averaging from all elements who share the node. This means if three faces share one node, the node is colored according to the average stress of the three. The colors are chosen as RGB values where red shades indicates positive stress (elongation) and blue shades indicate negative stress (compression).

It is important to note that at this stage the coloring of shell meshes is not fully functional, and may be very inaccurate. This is partly because of the definition of the local element axes. There has not been implemented any method to align the local axes in the same general direction. This means that in case of membrane stresses in x direction, some of the element may have the x direction pointing toward the global y direction, and thus the wrong value for some faces may be displayed. The Von Mises stress however has no general direction and includes all directions to find the "worst case", it also become strictly positive. The Von Mises is therefore more trustworthy than any specific direction, but does not differentiate between negative and positive stress, which decreases the value of the information. An example of how the Von Mises stresses in presented can be seen in Fig. 6.7.

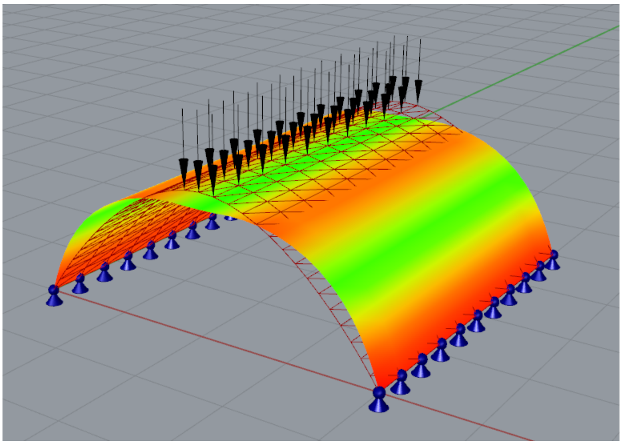


Figure 6.7: Von Mises stress color-map on structure

For well behaved meshes the local axes can however often be seen to coincide with each other and the global x direction. Some well behaved meshes can give remarkably consistently colored results, as in Fig. 6.8. It is important to note that fancy color distribution does not mean the results are correct in any way, this will be discussed further in Ch. 6.4.

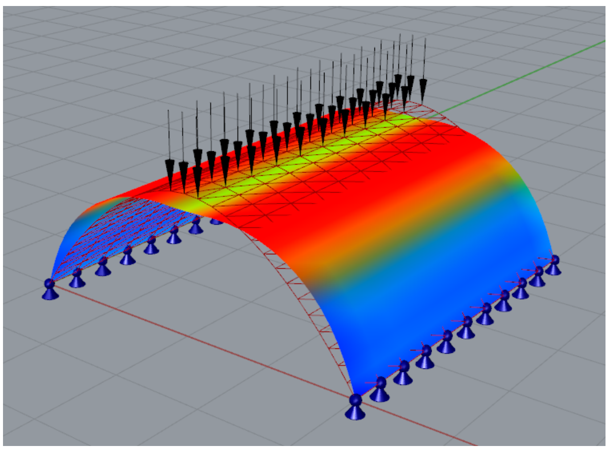


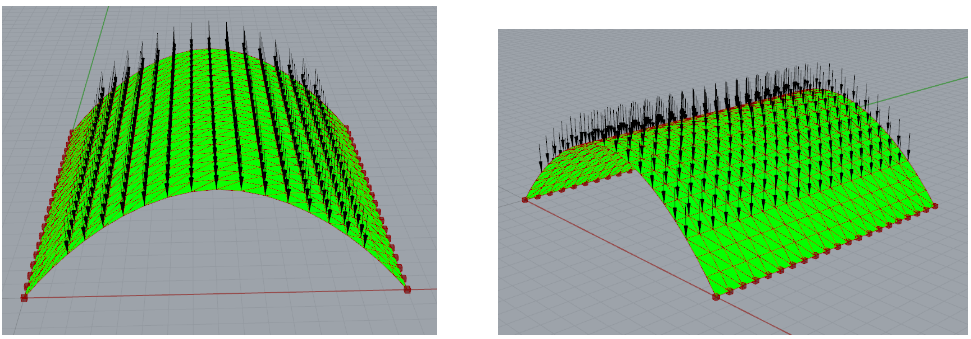
Figure 6.8: Well behaved local axes with membrane stress in x direction

6.3 Analysis

The following analyses has the focus on the main calculation component, the reasoning or this is discussed in Ch. 6.4. This software is aimed at real-time or hasty usage, therefore the two main parameters for usability is performance, which encompass the runtime of the shell software, and accuracy which indicate how close the results are to the ”actual” solution.

6.3.1 Performance

The performance analyses for the shell software naturally requires some example structures to analyze. Since the focus here will be on performance, the primary variable will be the number of elements. The first example structure will be referred to as the ”hangar”, and is shown in Fig. 6.9



(a) The hangar seen from the front (b) A slight side view of the hangar

Figure 6.9: The hangar example, dimensions 8 x 8 x 2.5 m

In Fig. 6.9 the nodes with arrows are loaded, and the sum of all the loads is 100 kN, regardless of how fine the mesh. This type of distributed load is a simple matter to make if one is familiar with Grasshopper. The boundary conditions are applied at the lower bounds of the structure, illustrated with ”fixed” boxes, where fixed means that the edges between the clamped nodes also are clamped in rotation.

To attain a sufficient overview of the time usage inside the calculation component, each of the steps in Fig. 6.3 has been timed. The components often vary slightly in runtime, therefore an average of five identical execution is used as the runtime for each part. The computation was carried out in 6 steps from 200 to 450 elements, the results can be seen in Fig. 6.10. The labels in the figure is clarified in Tab. 6.1.

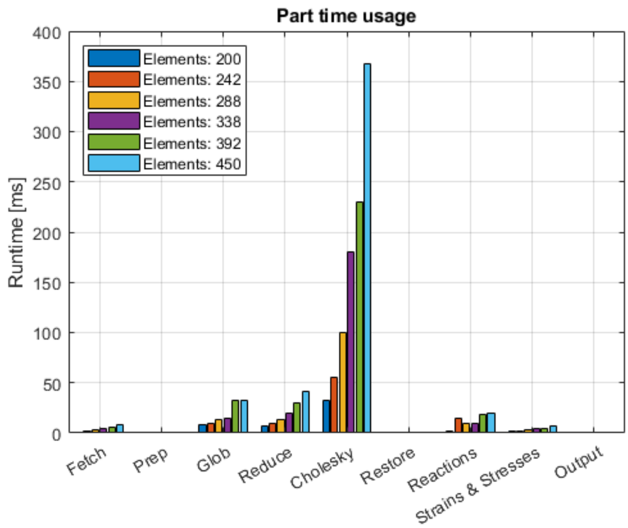


Figure 6.10: Runtime for the 9 steps in the main shell component

Table 6.1: Label clarification for analysis.

Name	**Description**
Fetch	Fetch Inputs
BDC & Load	Interpret loads and boundary conditions
El. & Glob.	Create element stiffness matrices and global stiffness matrix
Reduce	Reduce global stiffness matrix and load vector
Cholesky	Calculate reduced deformation vector using Cholesky
Restore	Restore total deformation vector
Reaction	Calculate reaction forces
S & S	Calculate internal strains and stresses
Output	Format output

Some small discrepancies can be noticed in Fig. 6.10 for the *El. & Glob.* and *Reaction*, which will be discussed further in Ch. 6.4.

In Ch. 6.1.1 it was mentioned that creating the reduced global stiffness matrix and load vector in a former version of the software was responsible for a noticeable part of the runtime. The average runtime for the old and new version of the *Reduce* part is shown in Fig. 6.11. The difference is mainly that the old method looped through the entire global stiffness matrix and the new only loops through the lower triangular part, as described in Ch. 6.1.1.

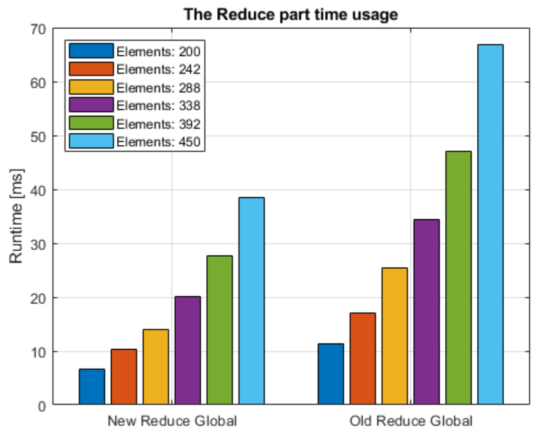


Figure 6.11: Comparison of the old and new reduction methods

The saved time in the new method might seem inconspicuous, and for the given number of elements it might be the case. However, for larger number of elements this might induce an noticeable undesired delay of the results.

The average total runtime of the calculation component can be extracted from Fig. 6.10 as the sum of all parts for each element count. The average total runtime for the component can be seen in Fig. 6.12.

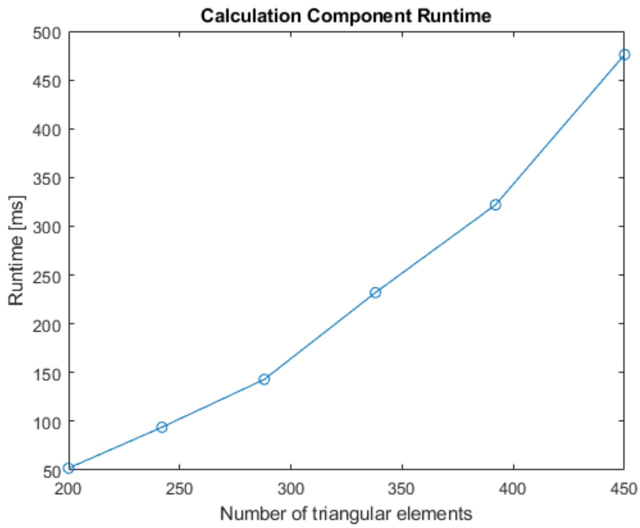


Figure 6.12: Hangar runtime of main component for 200 to 450 elements

It can be observed from Fig. 6.10 and Fig. 6.12 that, for these low numbers of elements, the component has some irregularities in the runtimes disturbs the expected exponential growth of the runtime curve. If the number of element is increased further up to 1152 elements as in Fig. 6.13, it can be noticed that the discrepancies does not have a very noticeable impact on the runtimes. The corresponding average calculation component runtime can also be seen in Fig. 6.14

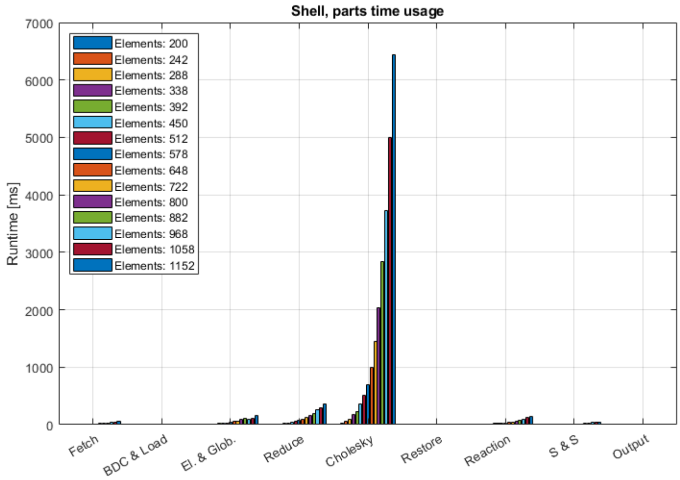


Figure 6.13: Hangar shell parts runtime for 200 to 1152 elements.

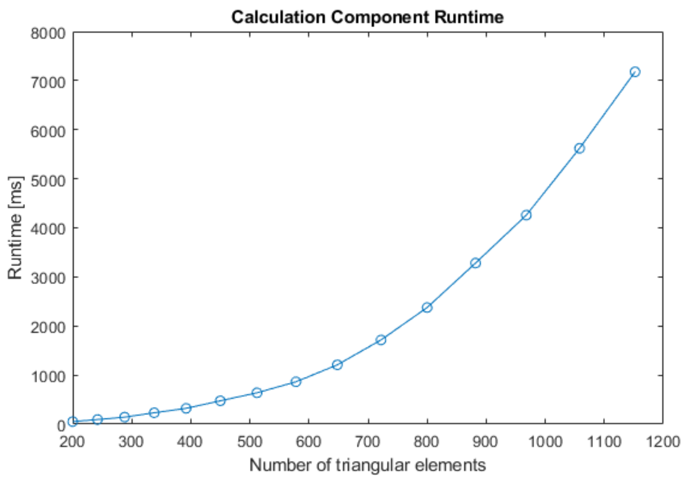


Figure 6.14: Hangar runtime of main comp. for 200 to 1152 elements

The expected exponential curve seems to be more apparent at this point. It can also be seen that the runtime for the calculation component with 1152 elements peaks just above seven seconds, which is quite noticeable when designing and updating the calculations for a structure.

In order to have more than just one structure to base all the performance results on, another example structure is introduced, namely the plate. The plate is shown in Fig. 6.15 and is located in the x-y plane for simplicity. It is loaded with a sum of 20 kN distributed over the mid area of the plate. The boundaries are fixed for translation and the connecting edges is fixed for rotation. Thus, the plate can be viewed as fixed at both edges which is symbolized with boxes in the figure.

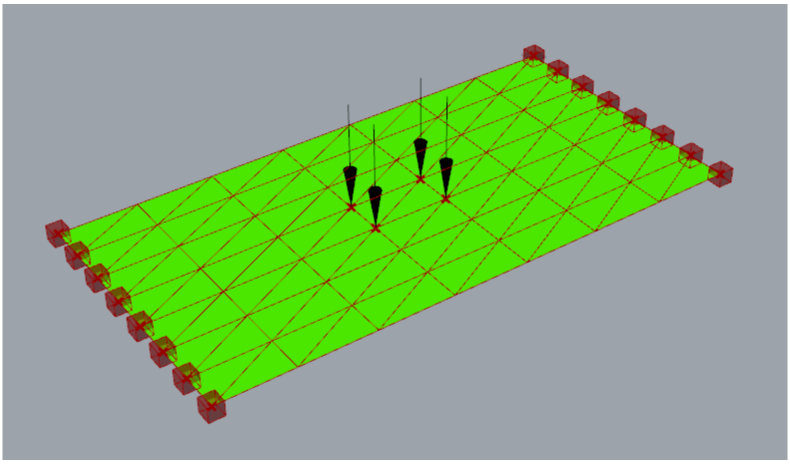


Figure 6.15: The Plate with dimensions 4 x 2 m

The calculations for the plate were also performed in steps from 200 to 1152 elements. The results can be previewed in Fig. 6.16, and the same pattern as in Fig. 6.13 seems to emerge. In fact if one plots the total component runtime for the two structures together as in Fig. 6.17, it is clear that they coincide very well and the differences is practically unnoticeable. The runtime does depend on both the number of element, but also the number of dofs. Which in the case of the hangar and the plate may be very much the same as they are relatively similarly structures and supported in a similar fashion.

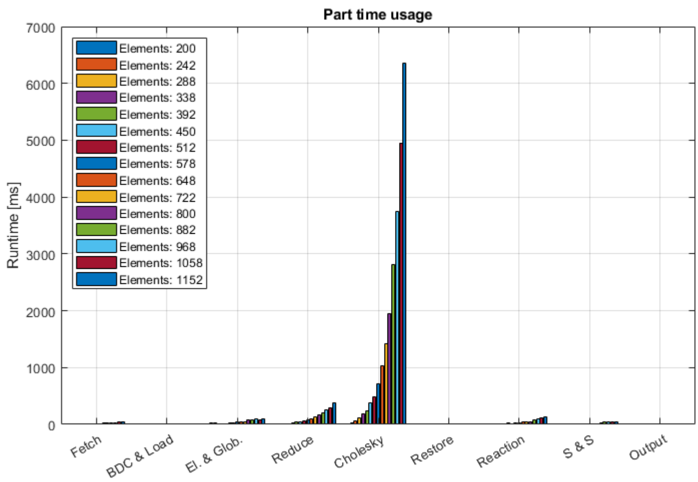


Figure 6.16: Plate main parts runtime for 200 to 1152 elements.

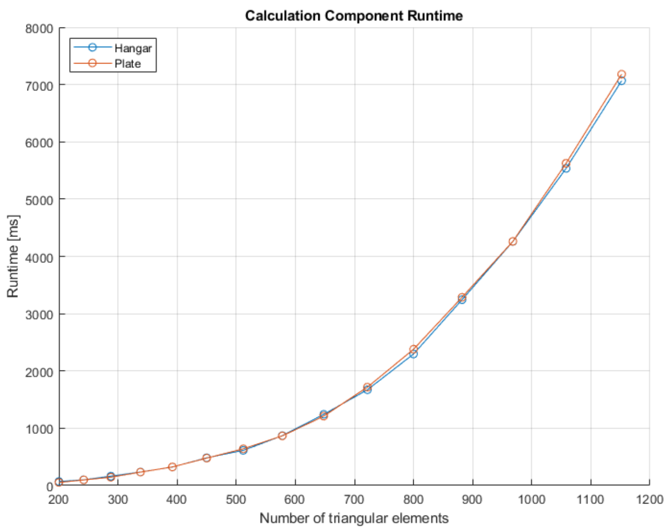


Figure 6.17: Hangar vs Plate runtime for 200 to 1152 elements.

Because of the similarity between the two previous examples another double curved shell will quickly be examined. The double curved shell structure in question is shown in Fig. 6.18. The double curved structure is also loaded with 100 kN divided over all the free nodes. Only four points are simply supported, and no rotations are restricted.

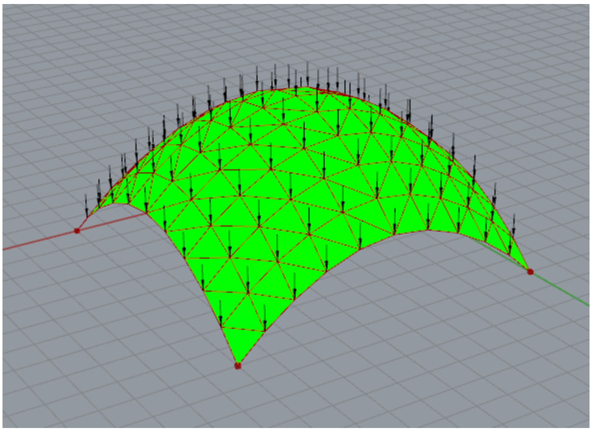


Figure 6.18: Double curved shell structure with dimensions 10 x 10 x 2.5m

The performance of the double curved structure would logically have a slightly higher runtime as the number of free dofs are greater than for the other two structures. The increased number dofs is the result of fewer nodes standing clamped. The difference in runtime is shown in Fig. 6.19, the difference is relatively beneath notice below roughly 500 elements, but becomes quite consequential when the runtime reaches several seconds.

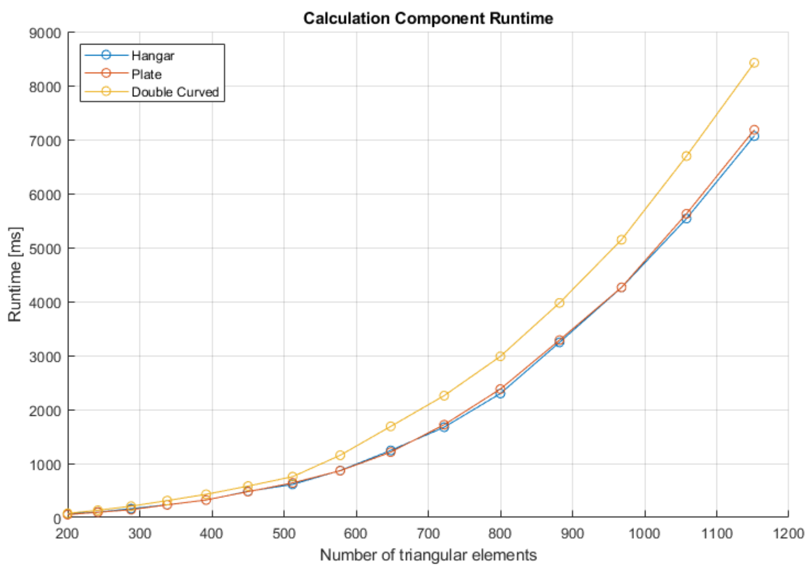


Figure 6.19: Comparison of component runtime for double curved shell

6.3.2 Accuracy

Considering that analytical solutions for shell structures are quite scarce and severely limited, and only simple examples can be analytically solved. For this reason, Autodesk Robot Structural Analysis software will be used as comparison for the results.

Firstly a simply supported plate will be examined. The plate can be analytically solved by Kirchhoff-Love plate theory as described in Ch. 2.6.3. The plate in question is a rectangular 4 by 2 meter plate with a constant distributed load, and is simply support along all edges. This means that no rotations is restrained but all translational dofs along the edge is clamped. The plate can be seen in Fig. 6.20, where the number of elements is very low for visual purposes.

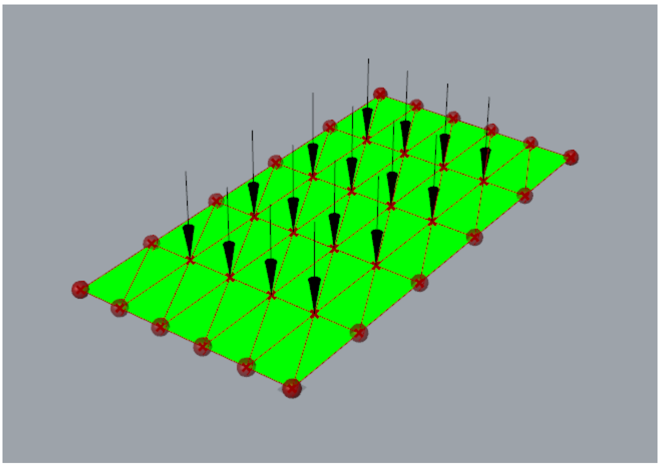


Figure 6.20: Plate to compare with analytical solution

A simply supported rectangular plate with a uniformly distributed load can be analytically solved for deformations by Navier's solution, which reads

$$w(x,y) = \frac{16q_0}{\pi^6 D} \sum_{m=1,3,5,..}^{\infty} \sum_{n=1,3,5,..}^{\infty} \frac{1}{mn(\frac{m^2}{a^2}+\frac{n^2}{b^2})^2} sin(\frac{m\pi x}{a}) sin(\frac{n\pi y}{b}) \quad \text{(Eq. 6.3.1)}$$

An important note is that the results from our created software is expected to converge towards the solution to be considered acceptable. The results from the made software is also preferred to be on the "safe side" of the solution to which it converges. In this software the "safe side" will be to get a larger deformation or higher stresses and strains than the "correct" solution. In the case of this plate the shell software solution will therefore hopefully give larger deformations than the analytical solution.

The focus for this accuracy test will be the midpoint of the plate, as this is the expected point for maximal deformation in negative z direction (downwards). The Navier solution has been implemented in Matlab, where the m and n variables had a maximum value of 1000. The solution from the Navier solution can be seen as the horizontal line in Fig. 6.21. The plate is initially set for 200 elements, and the results up to 2738 elements can be viewed in Fig. 6.21 below.

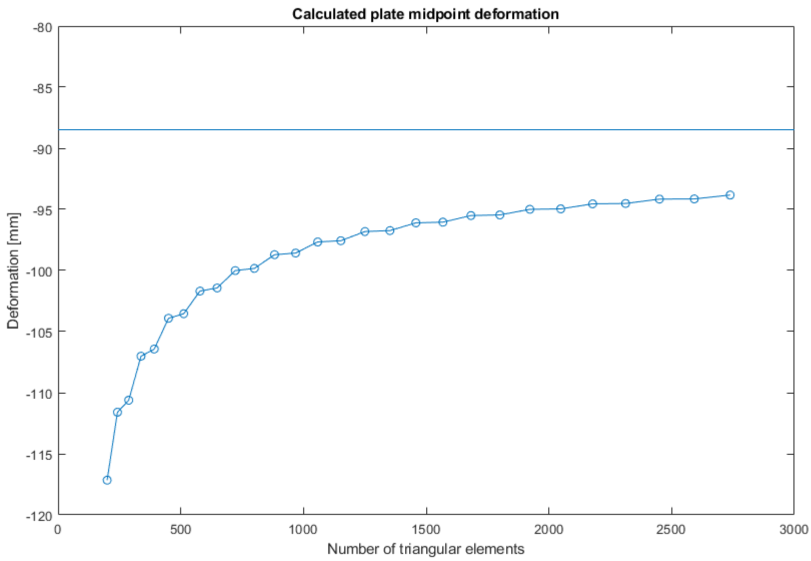


Figure 6.21: Deformation for the plate vs Navier's solution

From the figure it can be seen that our software gives a deformation that is worse than the analytical solution. It is also quite clear that as the element count increases the deformation converges towards the analytical solution. However, the element count grows quite large before the deformation approach Navier's solution for the plate.

The curve in Fig. 6.21 seem to form steps, this is as a result of the method used to refine the mesh. As the mesh is refined, it is simply split into a number in both x and y directions. These lines create squares which are then divided into triangles. The steps in the figure is a result of this refinement factor to be odd or even, where even numbers for refinement creates a node in the midpoint of the plate, while for odd numbers an edge will be in the midpoint. If an edge is at the midpoint of the plate the maximum deformation is "divided" between two nodes. This causes the even refinement factors to attain a slightly larger deformation as a single node appear in the point that has the most deformation in the plate.

The stresses in the plate can also be compared to that of the Navier solution. The corresponding equation for the maximal stress in the x direction σ_{xx} becomes

$$\sigma_{xx} = \frac{16hq_0}{2I_x\pi^4} \sum_{m=1,3,5,..}^{\infty} \sum_{n=1,3,5,..}^{\infty} \frac{\frac{n^2}{b^2} + \nu\frac{m^2}{a^2}}{mn(\frac{m^2}{a^2} + \frac{n^2}{b^2})^2} sin(\frac{m\pi x}{a}) sin(\frac{n\pi y}{b}) \quad \text{(Eq. 6.3.2)}$$

As this equation is solved with Matlab for the midpoint, the stresses from the shell software can now be plotted with the analytical solution as the target line. The plot can be seen in Fig. 6.22

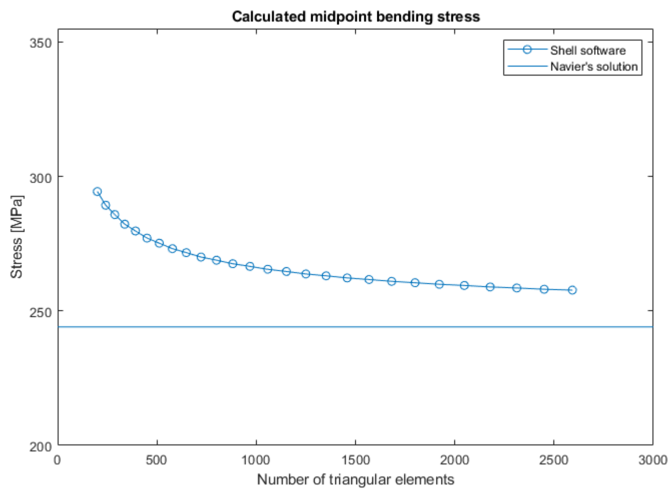


Figure 6.22: Maximum σ_{xx} stress for the plate vs Navier's solution

It can be seen from the figure that the stresses follow the same pattern as the deformation, and approaches the correct solution from the "safe side". It is clear from Fig. 6.22 that the stresses are not relatively far from the correct solution for the larger amount of elements.

The next structure to compare for accuracy will be a variation of the hangar from Ch. 6.3.1, which this time has the dimension 4 x 4 x 1.5 meters. To achieve the same loading a projected load of 6.25 kPa has been applied in robot, which over 4 x 4 meters gives a total of 100 kN. The structure in our shell software has been loaded with a total of 100 kN divided over the free nodes. This may not be entirely correct, but nevertheless is used as an approximation. Self-weight is not included in any of the software packages. A steel shell with a thickness of 15 mm and pinned support along the lower edges is set, and material parameters E = 210000 MPa and G = 80800 MPa has been chosen. The shell structure can be viewed in Fig. 6.23, with applied nodal loads and boundary conditions. The shell made in Grasshopper was exported to Robot to ensure that the same geometrical shape is used.

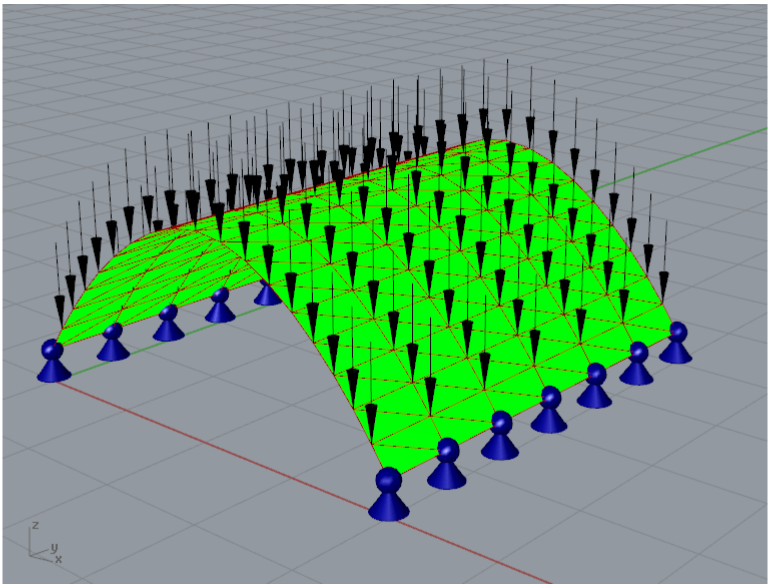


Figure 6.23: The generated shell from Grasshopper

The corresponding shell in Robot can be seen in Fig. 6.24. And the results from the calculation performed in robot can be seen in Tab. 6.2, and will be the approximate target values. The made software in grasshopper will hopefully also converge towards these results.

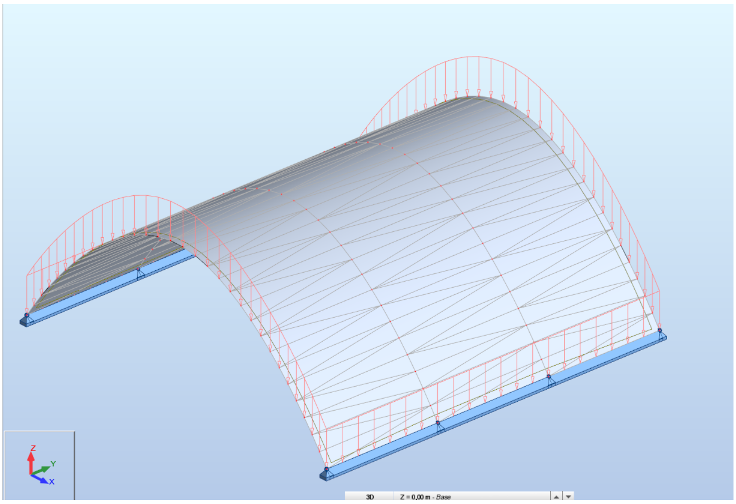


Figure 6.24: The generated shell from Robot

Table 6.2: Results from Robot calculation for the hangar structure.

Direction	**Max def.**	**Min def.**
X	0.4414 mm	-0.4220 mm
Z	0.4554 mm	-0.3610 mm

Principal σ_{max}	**Principal** σ_{min}
0.09 MPa	-1.04 MPa

A series of runs with varying number of elements gave the deformation in x direction as shown in Fig. 6.25, along with the deformations in z direction in Fig. 6.26. The deformation values are relatively much larger in the shell software than those from the Robot software. The deformation are on the ”safe side”, but they can be seen to be about twice as much or more. They do however converge towards the solution, but can, as seen from the figures, not be assumed to be sufficiently close for a practical amount of elements.

The stresses were also measured as the principal stress directions and are given in Fig. 6.27. The stresses can be observed to be extremely large compared to those from Robot. This is, among other factors, due to the error in the deformations as the stresses are calculated from Eq. 2.6.63 in combination with Eq. 2.6.30. Which makes the stresses directly dependent on the deformations, and when almost all deformations are larger than they should be, the cumulative effect results in amplified errors in the strains and therefore the stresses.

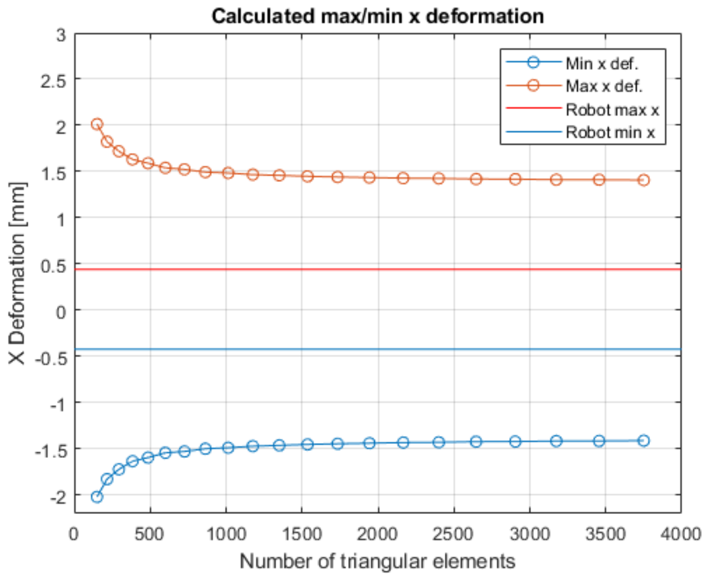


Figure 6.25: The measured x deformation for the hangar

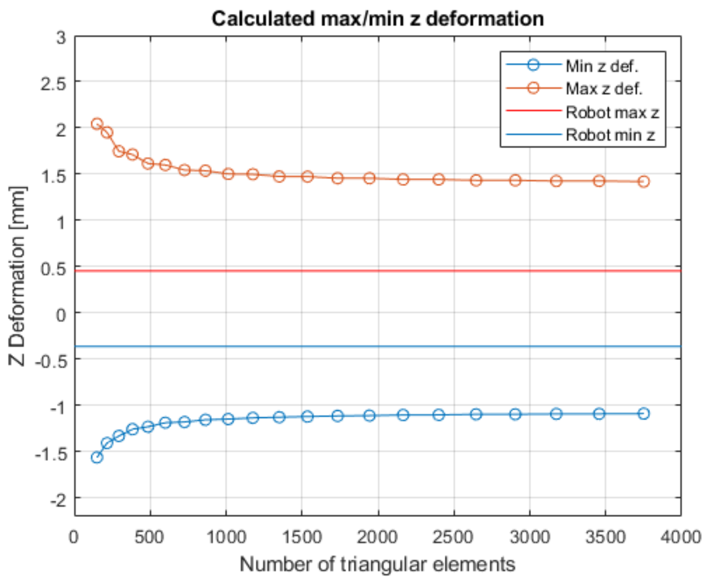


Figure 6.26: The measured z deformation for the hangar

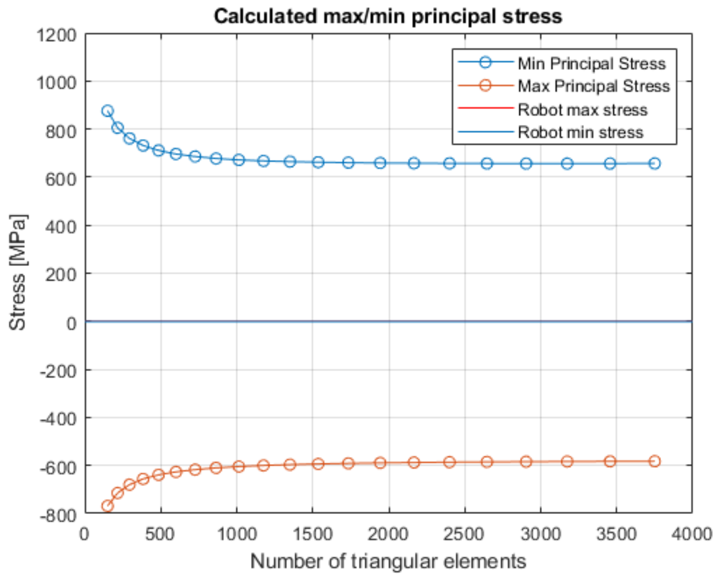


Figure 6.27: The measured principal stresses for the hangar

The stresses obtained from Robot can in Fig. 6.27 not be seen separately as they are so close due to the scale of the y axis. For steel, these values would be entirely incorrect as the shell would be far from yield with the given load, but according to our shell software it will yield. This is obviously an error of some sort and will be discussed in the next chapter.

The deformation shape however seems quite similar for the Shell software and Robot, the deformations from Robot can be seen in Fig. 6.28.

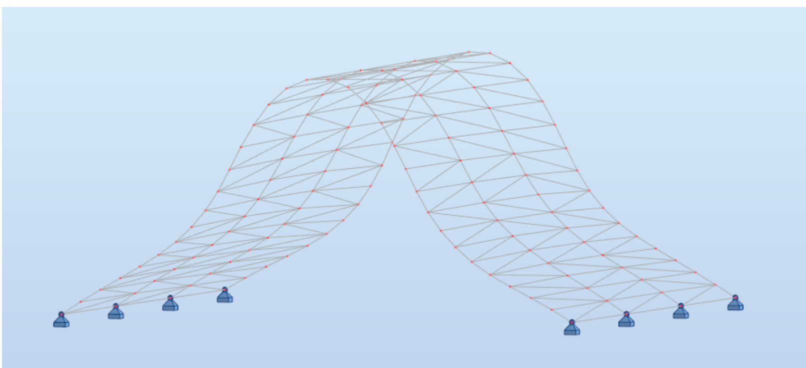


Figure 6.28: The deformation shape for the shell structure from Robot

And the deformations shape given by the shell software is shown in Fig. 6.29.

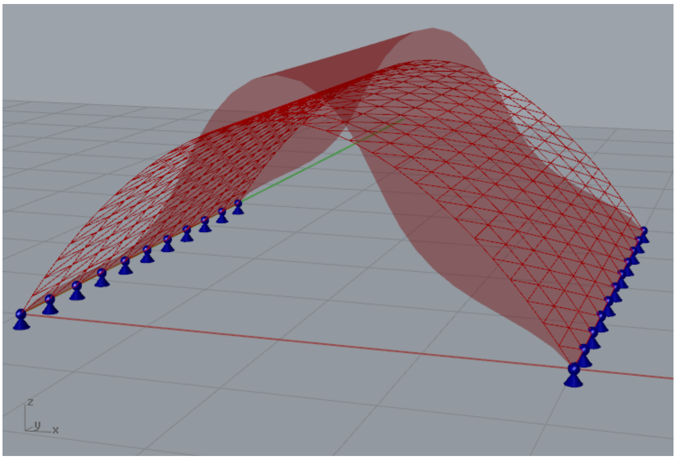


Figure 6.29: The deformation shape for the shell structure from Grasshopper

6.4 Discussion

The complexity in the shell software is noticeable higher than of the truss and beam, as a result of this the mishaps and bugs has proven a lot harder to locate. This made the shell software quite a bit harder and more time consuming to perfect.

In terms of calculation speed, most of the steps, see Fig. 6.3, of the main component can likely be sped up. But as of fig. 6.10 the runtime usage of the pre- and post-processing steps is relatively negligible compared to the processing. Even though the Cholesky solver was one of the most efficient solvers tested in this thesis, there are faster solvers, as for instance the ALGLIB package (ALGLIB, 2018). And as seen from the performance versus the accuracy, an increase in solving speed is needed. The solver may also be dependent on the structure of the matrix to solve for, and the global stiffness matrix for the shell software may be quite unfavorable if this is the case. This is a result of the decision to store all the rotational dofs at the end of the list of dofs, and therefore creating a matrix with a high spread. This could be corrected by locating the rotational dofs closer to the nodes they belong to, and this way make the global stiffness matrix more concentrated close to the diagonal. It is not certain if this will make any noticeable changes to the runtime, but it would be an interesting subject for further work.

The *DeformedGeometry* component for the shell software can be labeled as a work in progress, as it is capable of displaying color-maps for stresses, although this is not fully functional at this time. One of the underlying problems that is known is the orientation of the local axes of the elements. These are as of yet not oriented correctly for structures, but they can happen to be oriented correctly depending on the mesh and its face-orientation. This can be a very handy feature for designers to identify critical points, and should therefore be perfected in future work. The method Grasshopper uses for coloring can however be somewhat misleading if critical nodes are diluted by node averaging, this is also a problem to be investigated further in future work. There has however been found one other way of coloring meshes, this involves deconstructing the entire mesh and define each face as its own mesh, in this method node averaging would not be used, but the result might be quite chaotic and disconnected.

The shell software, at this point, aims to provide a relatively hasty solver for shell structures of low complexity. The meaning of low complexity in this context covers structures with few enough dofs and/or elements to be solved in a reasonable short time-span. It can however be used on more comprehensive structures, but is not built nor tested for such a purpose, and is therefore not recommended. For it to be a viable option in design it must not reduce the design process to mere than lingering for results. On the other hand, if the calculation is only limited by the runtime it may not be of much use as the results can be quite imprecise.

As seen from Fig. 6.10 the runtime of the pre- and post-processing steps in the main calculation component has a larger impact on the runtime for lower amounts of elements. This seems to only be noticeable beneath 500 elements, and even then, the variations is still negligible compared to the Cholesky solver.

For element counts below approximately 1000, the accuracy can be relatively unsatisfactory as seen in Fig. 6.21. This presents quite the predicament as the runtime for 1000 elements approaches five seconds according to Fig. 6.19. This may indicate that some major performance enhancement is due if the shell software is to be used for higher accuracy. However this software might still be of use as the deformation-patterns seems to be quite correct for all tested structures, as seen from Fig. 6.28 and 6.29. In this manner the software can be used with a relatively low number of elements while designing, and then be used for more detailed calculation with more elements only when needed. This could present an quicker way of approximating the behavior and locate critical areas in a structure, compared to handing it over to someone for assessment.

In terms of accuracy the shell software did relatively well compared to the analytical Navier solution of a Kirchhoff-Love plate. This might indicate that the handling of bending and the Morley triangle behaves and represent the plate deformations quite adequately. Still as seen in Fig. 6.25 - 6.27 there is something which does not work quite right in terms of accuracy. The culprit might be the CST triangle, which is only able to present constant stress and strain and therefore may be quite inaccurate for a low number of elements and rapidly changing stresses (Bell, 2013). As the results for displacement was about 300% of the expected value, there is reason to believe that there is still some unidentified mishap in the pre- or post-processing steps.

In terms of stresses the values from our shell software was quite extreme compared to that of those in Fig. 6.27. This might among else be the result of the inaccurate deformations, the poorly represented stress distribution and stress concentration. As our software is at an immature stage the stresses are not post-processed after being calculated which means it does not handle stress concentration close to supports, or even in the plane, in an good way. This is absolutely something that should be considered in future work.

Another inconsistency is the method of loading, as the loading in Grasshopper is done by dividing the total force over all free nodes, this might not be correct enough as the force on the edge elements are just as large as the force on all other nodes. In the case of evenly distributed loads with load lumping the force on the edge nodes is just half of the other nodes. In this manner the over loaded nodes might induce a larger deformation of the edges than the evenly distributed load in Robot. This could however be fixed by implementing a method for evenly and correctly distributing load, but as this is quite a time-consuming process it has not been prioritized as of yet. This could also be an interesting development for the software in future work.

On the positive side the obtained values from the analysis were all on the ”safe side”, some more than others. They were also seen to converge in the direction of the correct solutions, even if the values would not seem to be close in the foreseeable future. It is quite important for a finite element software to not be on the ”better” or ”unsafe side” as this would lead the user to believe thing are in order if they are not. In addition, it is imperative that the results are converging toward the correct solution, if not an increase of mesh resolution could give all kinds of wrong results.

The final but maybe obvious way to improve the software is the implementation of higher order elements. This could greatly improve the accuracy for lower amounts of elements, but would also include more dofs. On the same note it may not have been optimal to use the CST-Morley element in terms of accuracy, as it requires many elements in order to be somewhat precise. As observed in the analysis there might be some grave issues with the CST element in some structures and this could greatly benefit by being upgraded to e.g. a linear stress strain triangle to represent the stresses more accurately. The advancement of the element type could also include elements with four nodes, which could fit the way Grasshopper meshes structures even better than triangular elements do, but this may come with more work and problems than its worth.

The main calculation component could also benefit from being able to give even more kinds of data from the calculation and post-processing. This could be like the Von Mises stresses which is in this version of the software are given from the *DeformedGeometry* component.

6.5 Shell Summary

The shell software consists of four components, the main component is the calculation software and would be the core of the software. The other three are support component which support the main component by formatting boundary conditions, formatting loads and gives a preview of the deformed structure and the internal stresses through a color-map. Though the coloring is not fully functional as of yet, and needs some further work.

The software works surprisingly well in general and displays seemingly correct deformation patterns, however the deformations may not always be correct, depending on the structure type. A plate problem solved with Navier's analytical solution was compared and the software gave satisfactory results. Another hangar-like structure was compared against the solution from Robot Structural Analysis, and deviated very much. However, the deformation was still converging in direction of the correct result, and has not yet given lower results, which means it is on the ”safe side”.

The software need further work to be accurate enough, and still might not give the desired accuracy in an adequate runtime. As it stands now the software could seemingly be used to predict deformation patterns and to a limited extent give stress patterns. Which by itself could provide the user with some valuable information in the design process.

It seems a parametric FEA software for shells can be done, but the time usage might present some difficulties if good accuracy is sought. The created software for shell works in a parametric environment, and therefore the intention has been reach to a certain degree. However, some major improvements to the runtime and solving process can, and should, be undertaken to perfect the software in terms of runtime.

Chapter 7

Discussion

The reason for the support component to be excluded from the Analysis chapters is simply due to that the support components has a minuscule amount of operations to perform compared to the main calculation components. In this context it should be mentioned that Grasshopper's own timer, the *profiler* widget, would not even display the runtime for the support components. This indicates that they execute so fast it is not worth mentioning. The total runtime has also been perceived to heavily rely on the time usage of the main calculation components, and compared, the support components runtimes are negligible.

All the software packages assume identical material properties for all members. While support for individual properties would be a nice feature, the implementation of this has been assumed to be more time-consuming than worthwhile. The largest difficulties would presumably arise from organizing the various members and elements, which is outside the scope of this thesis.

On the same note, none of the packages have implemented self-weight loads. This is related to the lack of a proper solution for uniform load distributions. Some notes on how distributed loads could be implemented has briefly been mentioned in Ch. 5.4 and 6.4. To summarize, a fast but not entirely correct way to implement this could be through load-lumping, which would transform the distributed load into equivalent point-loads. This could easily be implemented, but has not been prioritized as it would be a time-consuming process.

As the different software packages has been analyzed, they have been compared to solutions from Robot Structural Analysis. This could be a somewhat imprecise comparison as Robot includes more advanced elements and structural effects which either has been neglected or simplified in these software packages. This in turn could impose some deviations in the results which cannot be closed. Through deeper and more time-consuming analyses of the structures in Robot, where these premises are accounted for, this gap could likely be remedied. However, this has not been prioritized since the focus has been on finding an approximate solution.

Among other goals for this thesis, one goal has been to attain some quick and approximate results which would indicate how the structure would react and deform, while give some pointers to the critical areas for stresses. The deformation part has been rather successful as all deformation shapes found so far has been very similar to the solutions from e.g. Robot. The results regarding stresses has been more troublesome than expected as the methods for visualizing the results has not been fully explored. This far the best solution seems to be the option to display stresses as color-maps on the structure, but this feature would benefit from more work and improvements. As mentioned in Ch. 6.4 the Shell software feature for coloring may not always be entirely correct for directional stresses, for stresses as Von Mises however, it gives some good pointers to the critical areas.

As first mentioned in Ch. 4.1.1, the Grasshopper interaction with C# proved problematic when it came to errors arising from incorrect node coordinates. Whether the problem stems from C# or Grasshopper is hard to say. Throughout the project, the double store format for numbers has been used rather than decimal, which has a higher accuracy. This may have been related to the issue, since the former can ”only” store up to 15 or 16 significant figures, while the latter is able to hold up to 28 or 29. However, this is unlikely to be the culprit, as most coordinates used in testing has been integers. Rounding of the coordinates is not much of an issue however, as the operation comes very cheaply, and the precision is still accurate at up to 10^{-5} mm.

The software packages all use a direct solution method, Cholesky Decomposition, when solving the systems of equations. For stable systems where speed is prioritized, Cholesky is a very efficient solver, albeit applicable to fewer problems than some alternatives (Bell, 2013). Cholesky being unable to solve matrices that are not positive-definite has been helpful more often than not, by indicating incorrect boundary conditions and other errors from preprocessing. An iterative solver like Jacobi or Gauss-Seidel would be beneficial in terms of memory usage, however, memory is rarely a problem unless working with especially large structures. Although useful, this has not been prioritized since most systems are likely to be within functional parameters for direct solving. A *general* recommendation from Poschmann et al. (1998) is to use direct solvers for 1D and 2D problems.

Employing a sparse matrix format such as the skyline matrix storage would also use less memory, and can be solved by Cholesky Decomposition for sparse matrices. This would be very useful since symbolic Cholesky factorization (algorithm for finding non-zero values) of a stiffness matrix can be reused even for different values (van Grondelle, 1999). Reusing information for factorization of A ($= \mathbf{K}$) is an incredibly convenient attribute in a parametric work environment, since models are expected to undergo numerous small changes. Note also that the values of A are independent of loads, meaning that the lower triangular matrix $\mathbf{L}$, and it's transposed $\mathbf{L}^{\mathbf{T}}$, are reusable for change in loading. Normally, direct solver methods are recommended for large number of load cases (Poschmann et al., 1998).

7.1 Further Work

If more time was available, it would have been worthwhile for this thesis to more deeply explore the possibilities around optimization of Cholesky. Since Math.NET does not support sparse matrix solvers, the Math.NET toolkit would likely be discarded in favor of e.g. ALGLIB (2018). Potentially, the solve algorithm could be built from scratch.

A topic for further work would be the combinations of the different software packages that has been made. The opportunity to combine different elements would greatly expand the capabilities of the software. However, this is complicated to implement since the packages are defined separately, and extensive groundwork would be required to facilitate this.

Adding support for orthotropic materials and varying thickness for shell, could be implemented without major changes. However, the issues concerning local axes directions discussed in Ch. 6.4 needs to be addressed for this to work correctly. The theoretical basis for both orthotropy and variable thickness are readily given in Ch. 2, and in the Shell software only need to be taken through the derivations to be implemented in the CST-Morley element. This could also open up for expansions as for materials like reinforced concrete, which may be varying in thickness and be anisotropic.

Chapter 8

Conclusion

Through this thesis, four parametric Finite Element Analysis (FEA) software packages have been created. The simplest were the 2D and 3D Truss which demonstrated great potential when compared to the well-established FEA software, Robot. The speed performance of the 2D and 3D Truss displayed great promise as running times were almost unnoticeable for the tested structures. The accuracy of the 3D Beam software was also relatively good in terms of accuracy compared to Robot, but could benefit from some improvement in running time for larger structures. The Shell software had diverse results on accuracy, with some being close to the analytical solution, but others being very distant. The Shell software would greatly benefit from a faster solver algorithm, as the running time for larger shell structures could quickly become very long.

The aim of providing a tool for a quick and rough assessment of a structure has been reached to some extent, but could benefit from further development in terms of accuracy and runtime. The software packages currently give a good indication of how the structure will deform linearly. Deformation shapes were found to coincide very well with the compared solutions from Robot Structural Analysis. There has also been implemented some coloring options to locate critical areas for stresses in shells. These has proven to work quite well for stresses independent of directions, as for instance Von Mises stress. Coloring of directional stresses is not fully functional as of yet, but does work for some structures. The other software packages do not have a component for coloring of stresses and strains, but this can be performed in Grasshopper by anyone experienced in the environment. The software packages can therefore be used as intended to assess early designs and structural behavior, naturally within some limitations.

In our opinion, the parametric environment of Grasshopper is well suited for implementation of light-weight FEA tools. However, the environment will to some degree limit how far the implementation and optimization of the FEA software can go. This is partially due to the limitations of meshing in Grasshopper, even though meshing options can probably be expanded by 3rd party components, much like our own. The foundation of Grasshopper and Rhino is made for designing rather than calculating. This is a good opportunity, since the design can be analyzed while designing, but it is also an impediment, since the foundation of Grasshopper and Rhino is not optimized for efficient calculations.

Our understanding of the aspects related to combining a parametric environment and a FEA software has been greatly expanded. During writing of this thesis, there have been challenges regarding efficient solving of linear systems of equations, organization of dofs, calculation of internal forces, visualization of results and much more. The parametric environment provides simple and flexible design opportunities and requires quick FEA to reach its potential. The running time has been found to be one of the main problems, but for a software whose main goal is to show the deformation shape and indicate critical areas, the runtime is usually satisfactory for design purposes.

Bibliography

Aalberg, A., November 2014. Lecture notes in tkt4122 - mechanics 2. Department of Structural Engineering, Norwegian University of Science and Technology.

ALGLIB, 2018. Numerical analysis library - dense solvers for linear systems, C++ and C# library.
URL `http://www.alglib.net/linear-solvers/dense.php`

Barber, J., 2011. Intermediate Mechanics of Materials. Springer Netherlands.

Bell, K., Jul. 2013. An engineering approach to FINITE ELEMENT ANALYSIS of linear structural mechanics problems. Akademika publishing.

Clough, R., Sep. 2001. Thoughts about the origin of the finite element method. Computers & Structures 79 (22-25), 2029–2030.

Gavin, H. P., 2012. Mathematical properties of stiffness matrices. Department of Civil and Environmental Engineering, Duke University.

K., G. K., L., M. J., November 1996. A brief history of the beginning of the finite element method. International Journal for Numerical Methods in Engineering 39 (22), 3761–3774.

Math.NET, 2018. Math.NET Numerics.
URL `https://numerics.mathdotnet.com/`

Mike A., Aug. 2016. Thin vs. Thick shells - Technical Knowledge Base - Computers and Structures, Inc. - Technical Knowledge Base.
URL `https://wiki.csiamerica.com/display/kb/Thin+vs.+Thick+shells`

Poschmann, P., Komzsik, L., Mayer, S., 06 1998. Direct or iterative? a dilemma for the user.

Ramsden, J., Aug. 2014. The mesh data structure in Grasshopper.
URL `http://james-ramsden.com/the-mesh-data-structure-in-grasshopper/`

Saouma, V., 1999. MATRIX STRUCTURAL ANALYSIS with an Introduction to Finite Elements. Dept. of Civil Environmental and Architectural Engineering University of Colorado,.

van Grondelle, J., August 1999. Symbolic sparse cholesky factorisation using elimination trees. Master's thesis, Utrecht University.

Williamson, Jr., F., November 1980. Richard courant and the finite element method: A further look. Historia Mathematica.

Appendix A

2D Truss

2D Truss Calculation Component

```
using System;
using System.Collections.Generic;
using System.Linq;
using Grasshopper.Kernel;
using Rhino.Geometry;
using TwoDTrussCalculation.Properties;

namespace TwoDTrussCalculation
{
    public class TwoDTrussCalculationComponent : GH_Component
    {
        public TwoDTrussCalculationComponent()
          : base("2D Truss Calc.", "2DTrussCalc",
              "Description",
              "Koala", "2D Truss")
        {
        }

        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddLineParameter("Lines", "LNS", "Geometry, in form
                of Lines)", GH_ParamAccess.list);
            pManager.AddTextParameter("Boundary Conditions", "BDC",
```

```
    "Boundary Conditions in form (x,z):1,1 where 1=free and 0=restrained", GH_ParamAccess.list);
            pManager.AddNumberParameter("Crossection area", "A", "Crossectional area, initial value 10e3 [mm*mm]", GH_ParamAccess.item, 10000);
            pManager.AddNumberParameter("Material E modulus", "E", "Material Property, initial value 200e3 [MPa]", GH_ParamAccess.item, 200000);
            pManager.AddTextParameter("Loads", "L", "Load given as Vector [N]", GH_ParamAccess.list);
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {

            pManager.AddNumberParameter("Deformations", "Def", "Deformations", GH_ParamAccess.list);
            pManager.AddNumberParameter("Reactions", "R", "Reaction Forces", GH_ParamAccess.list);
            pManager.AddNumberParameter("Element stresses", "Strs", "The Stress in each element", GH_ParamAccess.list);
            pManager.AddNumberParameter("Element strains", "Strn", "The Strain in each element", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            //Expected inputs
            List<Line> geometry = new List<Line>();         //initial Geometry of lines
            double E = 0;                                   //Material property, initial value 210000 [MPa]
            double A = 0;                                   //Area for each element in same order as geometry, initial value 10000 [mm^2]
            List<string> bdctxt = new List<string>();       //Boundary conditions in string format
            List<string> loadtxt = new List<string>();      //loads in string format


            //Set expected inputs from Indata
```

```
            if (!DA.GetDataList(0, geometry)) return;        //sets geometry
            if (!DA.GetDataList(1, bdctxt)) return;          //sets boundary conditions
            if (!DA.GetData(2, ref A)) return;               //sets Area
            if (!DA.GetData(3, ref E)) return;               //sets material
            if (!DA.GetDataList(4, loadtxt)) return;         //sets load


            //List all nodes (every node only once), numbering them according to list index
            List<Point3d> points = CreatePointList(geometry);


            //Interpret the BDC inputs (text) and create list of boundary condition (1/0 = free/clamped) for each dof.
            List<int> bdc_value = CreateBDCList(bdctxt, points);


            //Interpreting input load (text) and creating load list (double)
            List<double> load = CreateLoadList(loadtxt, points);


            //Create global stiffness matrix
            double[,] K_tot = CreateGlobalStiffnessMatrix(geometry, points, E, A);


            //Create the reduced global stiffness matrix and reduced load list
            int dofs_red = points.Count * 2 - (bdc_value.Count - bdc_value.Sum());                  //reduced number of dofs
            double[,] K_red = new double[dofs_red, dofs_red];            //preallocate reduced K matrix
            List<double> load_red = new List<double>();            //preallocate reduced load list
            CreateReducedGlobalStiffnessMatrix(points, bdc_value, K_tot, load, out K_red, out load_red);     //outputs are reduced K-matrix and reduced load list (removed free dofs)
```

```
            //Run the cholesky method for solving the system of equations
                for the deformations
            List<double> deformations_red = Cholesky_Banachiewicz(K_red,
                load_red);


            //Add the clamped dofs (= 0) to the deformations list
            List<double> deformations =
                RestoreTotalDeformationVector(deformations_red,
                bdc_value);


            //Calculate the reaction forces from the deformations
            List<double> Reactions =
                CalculateReactionforces(deformations, K_tot, bdc_value);


            //Calculate the internal strains and stresses in each member
            List<double> internalStresses;
            List<double> internalStrains;
            CalculateInternalStrainsAndStresses(deformations, points, E,
                geometry, out internalStresses, out internalStrains);

            //Set output data
            string K_print = PrintStiffnessMatrix(K_red);
            string K_print1 = PrintStiffnessMatrix(K_tot);

            DA.SetDataList(0, deformations);
            DA.SetDataList(1, Reactions);
            DA.SetDataList(2, internalStresses);
            DA.SetDataList(3, internalStrains);
        } //End of main program

        private void CalculateInternalStrainsAndStresses(List<double>
            def, List<Point3d> points, double E, List<Line> geometry, out
            List<double> internalStresses, out List<double>
            internalStrains)
        {
            //preallocating lists
            internalStresses = new List<double>(geometry.Count);
            internalStrains = new List<double>(geometry.Count);

```

```
            foreach (Line line in geometry)
            {
                int index1 = points.IndexOf(line.From);
                int index2 = points.IndexOf(line.To);

                //fetching deformation of point in x and y direction
                double u2 = def[index2 * 2];
                double v2 = def[index2 * 2 + 1];
                double u1 = def[index1 * 2];
                double v1 = def[index1 * 2 + 1];

                //creating new point at deformed coordinates
                double nx1 = points[index1].X + u1;
                double nz1 = points[index1].Z + v1;
                double nx2 = points[index2].X + u2;
                double nz2 = points[index2].Z + v2;

                //calculating dL = (length of deformed line - original length of line)
                double dL = Math.Sqrt(Math.Pow((nx2 - nx1), 2) + Math.Pow((nz2 - nz1), 2)) - line.Length;

                //calculating strain and stress
                internalStrains.Add(dL / line.Length);
                internalStresses.Add(internalStrains[internalStrains.Count - 1] * E);
            }
        }

        private List<double> RestoreTotalDeformationVector(List<double> deformations_red, List<int> bdc_value)
        {
            List<double> def = new List<double>();
            int index = 0;

            for (int i = 0; i < bdc_value.Count; i++)
            {
                if (bdc_value[i] == 0)
                {
                    def.Add(0);
                }
                else
                {
                    def.Add(deformations_red[index]);
```

```
                    index += 1;
                }
            }

            return def;
        }

        private List<double> CalculateReactionforces(List<double> def,
            double[,] K_tot, List<int> bdc_value)
        {
            List<double> R = new List<double>();

            for (int i = 0; i < K_tot.GetLength(1); i++)
            {
                if (bdc_value[i] == 0)
                {
                    double R_temp = 0;
                    for (int j = 0; j < K_tot.GetLength(0); j++)
                    {
                        R_temp += K_tot[i, j] * def[j];
                    }
                    R.Add(Math.Round(R_temp, 2));
                }
                else
                {
                    R.Add(0);
                }
            }
            return R;
        }

        private List<double> Cholesky_Banachiewicz(double[,] m,
            List<double> load)
        {
            double[,] A = m;
            List<double> load1 = load;

            //Cholesky only works for square, symmetric and positive
                definite matrices.
            //Square matrix is guaranteed because of how matrix is
                constructed, but symmetry is checked
            if (IsSymmetric(A))
            {
                //preallocating L and L_transposed matrices
```

```
            double[,] L = new double[m.GetLength(0), m.GetLength(1)];
            double[,] L_T = new double[m.GetLength(0),
                m.GetLength(1)];

            //creation of L and L_transposed matrices
            for (int i = 0; i < L.GetLength(0); i++)
            {
                for (int j = 0; j <= i; j++)
                {
                    double L_sum = 0;
                    if (i == j)
                    {
                        for (int k = 0; k < j; k++)
                        {
                            L_sum += L[i, k] * L[i, k];
                        }
                        L[i, i] = Math.Sqrt(A[i, j] - L_sum);
                        L_T[i, i] = L[i, i];
                    }
                    else
                    {
                        for (int k = 0; k < j; k++)
                        {
                            L_sum += L[i, k] * L[j, k];
                        }
                        L[i, j] = (1 / L[j, j]) * (A[i, j] - L_sum);
                        L_T[j, i] = L[i, j];
                    }
                }
            }
            //Solving L*y=load1 for temporary variable y
            List<double> y = ForwardsSubstitution(load1, L);


            //Solving L^T*x = y for deformations x
            List<double> x = BackwardsSubstitution(load1, L_T, y);

            return x;
        }
        else    //K-matrix is not symmetric
        {
            //throw new RuntimeException("Matrix is not symmetric");
            System.Diagnostics.Debug.WriteLine("Matrix is not
                symmetric (ERROR!)");
```

```
                return null;
            }
        }

        private List<double> ForwardsSubstitution(List<double> load1,
            double[,] L)
        {
            List<double> y = new List<double>();
            for (int i = 0; i < L.GetLength(1); i++)
            {
                double L_prev = 0;

                for (int j = 0; j < i; j++)
                {
                    L_prev += L[i, j] * y[j];
                }
                y.Add((load1[i] - L_prev) / L[i, i]);
            }
            return y;
        }

        private List<double> BackwardsSubstitution(List<double> load1,
            double[,] L_T, List<double> y)
        {
            var x = new List<double>(new double[load1.Count]);
            for (int i = L_T.GetLength(1) - 1; i > -1; i--)
            {
                double L_prev = 0;

                for (int j = L_T.GetLength(1) - 1; j > i; j--)
                {
                    L_prev += L_T[i, j] * x[j];
                }

                x[i] = ((y[i] - L_prev) / L_T[i, i]);
            }
            return x;
        }

        private static void
            CreateReducedGlobalStiffnessMatrix(List<Point3d> points,
            List<int> bdc_value, double[,] K_tot, List<double> load, out
            double[,] K_red, out List<double> load_red)
        {
```

```
            int dofs_red = points.Count * 2 - (bdc_value.Count -
                bdc_value.Sum());
            double[,] K_redu = new double[dofs_red, dofs_red];
            List<double> load_redu = new List<double>();
            List<int> bdc_red = new List<int>();
            int m = 0;
            for (int i = 0; i < K_tot.GetLength(0); i++)
            {
                if (bdc_value[i] == 1)
                {
                    int n = 0;
                    for (int j = 0; j < K_tot.GetLength(1); j++)
                    {
                        if (bdc_value[j] == 1)
                        {
                            K_redu[m, n] = K_tot[i, j];
                            n++;
                        }
                    }

                    load_redu.Add(load[i]);

                    m++;
                }
            }
            load_red = load_redu;
            K_red = K_redu;
        }

        private double[,] CreateGlobalStiffnessMatrix(List<Line>
            geometry, List<Point3d> points, double E, double A)
        {
            int dofs = points.Count * 2;
            double[,] K_tot = new double[dofs, dofs];

            for (int i = 0; i < geometry.Count; i++)
            {
                Line currentLine = geometry[i];
                double mat = (E * A) / (currentLine.Length);
                Point3d p1 = currentLine.From;
                Point3d p2 = currentLine.To;

                double angle = Math.Atan2(p2.Z - p1.Z, p2.X - p1.X);
                double c = Math.Cos(angle);
```

```
                double s = Math.Sin(angle);

                double[,] K_elem = new double[,]{
                    { c* c* mat, s*c* mat, -c*c* mat, -s * c* mat},
                    { s* c* mat, s*s* mat, -s * c* mat, -s*s* mat},
                    { -c*c* mat, -s * c* mat, c* c* mat, s*c* mat},
                    { -s* c* mat, -s* s* mat, s* c* mat, s*s* mat} };

                int node1 = points.IndexOf(p1);
                int node2 = points.IndexOf(p2);

                //upper left corner of k-matrix
                K_tot[node1 * 2, node1 * 2] += K_elem[0, 0];
                K_tot[node1 * 2, node1 * 2 + 1] += K_elem[0, 1];
                K_tot[node1 * 2 + 1, node1 * 2] += K_elem[1, 0];
                K_tot[node1 * 2 + 1, node1 * 2 + 1] += K_elem[1, 1];

                //upper right corner of k-matrix
                K_tot[node1 * 2, node2 * 2] += K_elem[0, 2];
                K_tot[node1 * 2, node2 * 2 + 1] += K_elem[0, 3];
                K_tot[node1 * 2 + 1, node2 * 2] += K_elem[1, 2];
                K_tot[node1 * 2 + 1, node2 * 2 + 1] += K_elem[1, 3];

                //lower left corner of k-matrix
                K_tot[node2 * 2, node1 * 2] += K_elem[2, 0];
                K_tot[node2 * 2, node1 * 2 + 1] += K_elem[2, 1];
                K_tot[node2 * 2 + 1, node1 * 2] += K_elem[3, 0];
                K_tot[node2 * 2 + 1, node1 * 2 + 1] += K_elem[3, 1];

                //lower right corner of k-matrix
                K_tot[node2 * 2, node2 * 2] += K_elem[2, 2];
                K_tot[node2 * 2, node2 * 2 + 1] += K_elem[2, 3];
                K_tot[node2 * 2 + 1, node2 * 2] += K_elem[3, 2];
                K_tot[node2 * 2 + 1, node2 * 2 + 1] += K_elem[3, 3];
            }

            return K_tot;
        }

        private List<double> CreateLoadList(List<string> loadtxt,
            List<Point3d> points)
        {
            List<double> loads = new List<double>();
            List<double> inputLoads = new List<double>();
```

```
            List<double> coordlist = new List<double>();

            for (int i = 0; i < loadtxt.Count; i++)
            {
                string coordstr = (loadtxt[i].Split(':')[0]);
                string loadstr = (loadtxt[i].Split(':')[1]);

                string[] coordstr1 = (coordstr.Split(','));
                string[] loadstr1 = (loadstr.Split(','));

                inputLoads.Add(Math.Round(double.Parse(loadstr1[0])));
                inputLoads.Add(Math.Round(double.Parse(loadstr1[1])));
                inputLoads.Add(Math.Round(double.Parse(loadstr1[2])));

                coordlist.Add(Math.Round(double.Parse(coordstr1[0])));
                coordlist.Add(Math.Round(double.Parse(coordstr1[1])));
                coordlist.Add(Math.Round(double.Parse(coordstr1[2])));
            }

            int loadIndex = 0; //bdc_points index

            for (int i = 0; i < points.Count; i++)
            {

                double cptx = Math.Round(points[i].X);
                double cpty = Math.Round(points[i].Y);
                double cptz = Math.Round(points[i].Z);
                bool foundPoint = false;

                for (int j = 0; j < coordlist.Count / 3; j++) if
                    (loadIndex < coordlist.Count)
                    {
                        if (coordlist[j * 3] == cptx && coordlist[j * 3 +
                            1] == cpty && coordlist[j * 3 + 2] == cptz)
                        {
                            loads.Add(inputLoads[loadIndex]);
                            loads.Add(inputLoads[loadIndex + 2]);
                            loadIndex += 3;
                            foundPoint = true;
                        }
                    }

                if (foundPoint == false)
                {
```

```
                    loads.Add(0);
                    loads.Add(0);
                }
            }


            return loads;
        }

        private List<int> CreateBDCList(List<string> bdctxt,
            List<Point3d> points)
        {
            List<int> bdc_value = new List<int>();
            List<int> bdcs = new List<int>();
            List<double> bdc_points = new List<double>(); //Coordinates
                relating til bdc_value in for (eg. x y z)
            int bdcIndex = 0; //bdc_points index

            for (int i = 0; i < bdctxt.Count; i++)
            {
                string coordstr = (bdctxt[i].Split(':')[0]);
                string bdcstr = (bdctxt[i].Split(':')[1]);

                string[] coordstr1 = (coordstr.Split(','));
                string[] bdcstr1 = (bdcstr.Split(','));

                bdc_points.Add(double.Parse(coordstr1[0]));
                bdc_points.Add(double.Parse(coordstr1[1]));
                bdc_points.Add(double.Parse(coordstr1[2]));

                bdcs.Add(int.Parse(bdcstr1[0]));
                bdcs.Add(int.Parse(bdcstr1[1]));
                bdcs.Add(int.Parse(bdcstr1[2]));
            }

            for (int i = 0; i < points.Count; i++)
            {

                double cptx = points[i].X;
                double cpty = points[i].Y;
                double cptz = points[i].Z;
                bool foundPoint = false;

                for (int j = 0; j < bdc_points.Count / 3; j++) if
```

```
                    (bdcIndex < bdc_points.Count)
                    {
                        if (bdc_points[bdcIndex] == cptx &&
                            bdc_points[bdcIndex + 1] == cpty &&
                            bdc_points[bdcIndex + 2] == cptz)
                        {
                            bdc_value.Add(bdcs[bdcIndex]);
                            bdc_value.Add(bdcs[bdcIndex + 2]);
                            bdcIndex += 3;
                            foundPoint = true;
                        }
                    }

                if (foundPoint == false)
                {
                    bdc_value.Add(1);
                    bdc_value.Add(1);
                }
            }

            return bdc_value;
        }

        private List<Point3d> CreatePointList(List<Line> geometry)
        {
            List<Point3d> points = new List<Point3d>();

            for (int i = 0; i < geometry.Count; i++) //adds every point unless it already exists in list
            {
                Line l1 = geometry[i];
                if (!points.Contains(l1.From))
                {
                    points.Add(l1.From);
                }
                if (!points.Contains(l1.To))
                {
                    points.Add(l1.To);
                }
            }

            return points;
        }

```

```
        private static bool IsSymmetric(double[,] A)
        {
            int rowCount = A.GetLength(0);
            for (int i = 0; i < rowCount; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    if (A[i, j] != A[j, i])
                    {
                        return false;
                    }
                }
            }
            return true;
        }

        public override GH_Exposure Exposure
        {
            get { return GH_Exposure.primary; }
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Resources.TwoDTrussCalculation;    //Setting component icon
            }
        }

        public override Guid ComponentGuid
        {
            get { return new Guid("beae0421-b363-41de-89a2-49cca8210736"); }
        }
    }
}
```

2D Truss Point Loads Component

```
using System;
using System.Collections.Generic;
using Grasshopper.Kernel;
using Rhino.Geometry;

namespace TwoDTrussCalculation
{
    public class Point_Load : GH_Component
    {

        public Point_Load()
          : base("PointLoads", "PL",
            "Set one or more pointloads on nodes",
            "Koala", "2D Truss")
        {
        }

        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply
                load(s)", GH_ParamAccess.list);
            pManager.AddNumberParameter("Load", "L", "Load magnitude
                [Newtons]. Give either one load to be applied to all
                inputted points, or different loads for each inputted
                loads", GH_ParamAccess.list);
            pManager.AddNumberParameter("angle (xz)", "a", "Angle
                [degrees] for load in xz plane", GH_ParamAccess.list, 90);
            //pManager[2].Optional = true; //Code can run without a given
                angle (90 degrees is initial value)
        }

        protected override void
            RegisterOutputParams(GH_Component.GH_OutputParamManager
            pManager)
        {
            pManager.AddTextParameter("PointLoads", "PL", "PointLoads
                formatted for Truss Calculation", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
```

```
    {
        //Expected inputs and output
        List<Point3d> pointList = new List<Point3d>();
            //List of points where load will be applied
        List<double> loadList = new List<double>();
        List<double> anglexz = new List<double>();
            //Initial xz angle 90
        List<double> anglexy = new List<double> { 0 };
            //Initial xy angle  0
        List<string> pointInStringFormat = new List<string>();
            //preallocate final string output

        //Set expected inputs from Indata
        if (!DA.GetDataList(0, pointList)) return;
        if (!DA.GetDataList(1, loadList)) return;
        DA.GetDataList(2, anglexz);

        //initialize temporary stringline and load vectors
        string vectorString;
        double load = 0;
        double xvec = 0;
        double yvec = 0;
        double zvec = 0;

        if (loadList.Count == 1 && anglexz.Count == 1)
            //loads and angles are identical for all points
        {
            load = -1 * loadList[0];
                //negativ load for z-dir
            xvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI /
                180) * Math.Cos(anglexy[0] * Math.PI / 180), 2);
            yvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI /
                180) * Math.Sin(anglexy[0] * Math.PI / 180), 2);
            zvec = Math.Round(load * Math.Sin(anglexz[0] * Math.PI /
                180), 2);

            vectorString = xvec + "," + yvec + "," + zvec;
            for (int i = 0; i < pointList.Count; i++)
                //adds identical load to all points in pointList
            {
                pointInStringFormat.Add(pointList[i].X + "," +
                    pointList[i].Y + "," + pointList[i].Z + ":" +
                    vectorString);
            }
```

```
            }
            else    //loads and angles may be different => calculate new xvec, yvec, zvec for all loads
            {
                for (int i = 0; i < pointList.Count; i++)
                {
                    if (loadList.Count < i)              //if pointlist is larger than loadlist, set last load value in remaining points
                    {
                        vectorString = xvec + "," + yvec + "," + zvec;
                    }
                    else
                    {
                        load = -1 * loadList[i];         //negative load for z-dir

                        xvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Cos(anglexy[i]), 2);
                        yvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Sin(anglexy[i]), 2);
                        zvec = Math.Round(load * Math.Sin(anglexz[i]), 2);

                        vectorString = xvec + "," + yvec + "," + zvec;
                    }

                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + vectorString);
                }
            }

            //Set output data
            DA.SetDataList(0, pointInStringFormat);
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                // You can add image files to your project resources and access them like this:
                //return Resources.IconForThisComponent;
                return Properties.Resources.PointLoad;
```

```
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("f6167454-39ae-4204-bfde-0254a1dc6578"); }
        }
    }
}
```

2D Truss BDC Component

```
using System;
using System.Collections.Generic;
using Grasshopper.Kernel;
using Rhino.Geometry;
using TwoDTrussCalculation.Properties;

namespace TwoDTrussCalculation
{
    public class BoundaryConditions : GH_Component
    {

        public BoundaryConditions()
          : base("BDC", "BDC",
              "Set boundary conditions at nodes",
              "Koala", "2D Truss")
        {
        }

        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply
                Boundary Conditions", GH_ParamAccess.list);
            pManager.AddIntegerParameter("Boundary Conditions", "BDC",
                "Boundary Conditions x,y,z where 0=clamped and 1=free",
                GH_ParamAccess.list, new List<int>(new int[] { 0, 0, 0
                }));
        }

        protected override void
            RegisterOutputParams(GH_Component.GH_OutputParamManager
            pManager)
        {
            pManager.AddTextParameter("B.Cond.", "BDC", "Boundary
                Conditions for 2D Truss Calculation",
                GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            //Expected inputs
```

```
            List<Point3d> pointList = new List<Point3d>(); //List of points where BDC is to be applied
            List<int> BDC = new List<int>();                       //is BDC free? (=clamped) (1 == true, 0 == false)
            List<string> pointInStringFormat = new List<string>(); //output in form of list of strings


            //Set expected inputs from Indata and aborts with error message if input is incorrect
            if (!DA.GetDataList(0, pointList)) return;
            if (!DA.GetDataList(1, BDC)) { AddRuntimeMessage(GH_RuntimeMessageLevel.Warning, "testing"); return; }


            //Preallocate temporary variables
            string BDCString;
            int bdcx = 0;
            int bdcy = 0;
            int bdcz = 0;


            if (BDC.Count == 3) //Boundary condition input for identical conditions in all points. Split into if/else for optimization
            {
                bdcx = BDC[0];
                bdcy = BDC[1];
                bdcz = BDC[2];

                BDCString = bdcx + "," + bdcy + "," + bdcz;

                for (int i = 0; i < pointList.Count; i++)   //Format stringline for all points (identical boundary conditions for all points)
                {
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);
                }
            }
            else    //BDCs are not identical for all points
            {
```

```
                for (int i = 0; i < pointList.Count; i++)
                {
                    if (i > (BDC.Count / 3) - 1)  //Are there more points
                        than BDCs given? (BDC always lists x,y,z per
                        point)
                    {
                        BDCString = bdcx + "," + bdcy + "," + bdcz; //use
                            values from last BDC in list of BDCs
                    }
                    else
                    {
                        //retrieve BDC for x,y,z-dir
                        bdcx = BDC[i * 3];
                        bdcy = BDC[i * 3 + 1];
                        bdcz = BDC[i * 3 + 2];
                        BDCString = bdcx + "," + bdcy + "," + bdcz;
                    }
                    pointInStringFormat.Add(pointList[i].X + "," +
                        pointList[i].Y + "," + pointList[i].Z + ":" +
                        BDCString);    //Add stringline to list of strings
                }
            }
            DA.SetDataList(0, pointInStringFormat);
        } //End of main program

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Resources.BoundaryCondition; //Setting component
                    icon
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("0efc7b95-936a-4c88-8005-485398c61a31"); }
        }
    }
}
```

2D Truss Deformed Geometry Component

```
using System;
using System.Collections.Generic;
using Grasshopper.Kernel;
using Rhino.Geometry;
using TwoDTrussCalculation.Properties;

namespace TwoDTrussCalculation
{
    public class DrawDeformedGeometry : GH_Component
    {
        /// <summary>
        /// Initializes a new instance of the DrawDeformedGeometry class.
        /// </summary>
        public DrawDeformedGeometry()
          : base("Def.Geom.", "Def.Geom.",
              "Displays the deformed geometry based on given deformations",
              "Koala", "2D Truss")
        {
        }

        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddNumberParameter("Deformation", "Def", "The Node Deformation from 2DTrussCalc", GH_ParamAccess.list);
            pManager.AddLineParameter("Geometry", "G", "Input Geometry (Line format)", GH_ParamAccess.list);
            pManager.AddNumberParameter("Scale", "S", "The Scale Factor for Deformation", GH_ParamAccess.item);
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddLineParameter("Deformed Geometry", "Def.G.", "Deformed Geometry as List of Lines", GH_ParamAccess.list);
        }

```

```
        protected override void SolveInstance(IGH_DataAccess DA)
        {
            //Expected inputs and outputs
            List<double> def = new List<double>();
            List<Line> geometry = new List<Line>();
            double scale = 0;
            List<Line> defGeometry = new List<Line>();
            List<Point3d> defPoints = new List<Point3d>();

            //Set expected inputs from Indata
            if (!DA.GetDataList(0, def)) return;
            if (!DA.GetDataList(1, geometry)) return;
            if (!DA.GetData(2, ref scale)) return;

            //List all nodes (every node only once), numbering them
                according to list index
            List<Point3d> points = CreatePointList(geometry);

            int index = 0;
            //loops through all points and scales x- and z-dir
            foreach (Point3d point in points)
            {
                //fetch global x,y,z placement of point
                double x = point.X;
                double y = point.Y;
                double z = point.Z;

                //scales x and z according to input Scale (ignores y-dir
                    since 2D)
                defPoints.Add(new Point3d(x + scale * def[index], y, z +
                    scale * def[index + 1]));
                index += 2;
            }

            //creates deformed geometry based on initial geometry
                placement
            foreach (Line line in geometry)
            {
                //fetches index of original start and endpoint
                int i1 = points.IndexOf(line.From);
                int i2 = points.IndexOf(line.To);

                //creates new line based on scaled deformation of said
                    points
```

```
                defGeometry.Add(new Line(defPoints[i1], defPoints[i2]));
            }


            //Set output data
            DA.SetDataList(0, defGeometry);
        }   //End of main program

        private List<Point3d> CreatePointList(List<Line> geometry)
        {
            List<Point3d> points = new List<Point3d>();

            for (int i = 0; i < geometry.Count; i++) //adds every point
                unless it already exists in list
            {
                Line l1 = geometry[i];
                if (!points.Contains(l1.From))
                {
                    points.Add(l1.From);
                }
                if (!points.Contains(l1.To))
                {
                    points.Add(l1.To);
                }
            }

            return points;
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Resources.DrawDeformedGeometry;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("bc7b48e4-4234-4420-bd7a-5a59220aba67"); }
        }
    }
}
```

Appendix B

3D Truss

3D Truss calculation Component

```
using System;
using System.Collections.Generic;
using Grasshopper.Kernel;
using Rhino.Geometry;
using MathNet.Numerics.LinearAlgebra;
using MathNet.Numerics.LinearAlgebra.Double;

namespace Truss3D
{
    public class CalcComponent : GH_Component
    {
        public CalcComponent()
          : base("Truss3DCalc", "TCalc",
              "Description",
              "Koala", "Truss3D")
        {
        }

        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddLineParameter("Lines", "LNS", "Geometry, in form
                of Lines)", GH_ParamAccess.list);
            pManager.AddTextParameter("Boundary Conditions", "BDC",
```

```
            "Boundary Conditions in form (x,z):1,1 where 1=free and
            0=restrained", GH_ParamAccess.list);
        pManager.AddNumberParameter("Crossection area", "A",
            "Crossectional area, initial value 3600 [mm^2]",
            GH_ParamAccess.item, 3600);
        pManager.AddNumberParameter("Material E modulus", "E",
            "Material Property, initial value 200e3 [MPa]",
            GH_ParamAccess.item, 200000);
        pManager.AddTextParameter("Loads", "L", "Load given as Vector
            [N]", GH_ParamAccess.list);
    }

    protected override void
        RegisterOutputParams(GH_Component.GH_OutputParamManager
        pManager)
    {
        pManager.AddNumberParameter("Deformations", "Def",
            "Deformations", GH_ParamAccess.list);
        pManager.AddNumberParameter("Reactions", "R", "Reaction
            Forces", GH_ParamAccess.list);
        pManager.AddNumberParameter("Element stresses", "Strs", "The
            Stress in each element", GH_ParamAccess.list);
        pManager.AddNumberParameter("Element strains", "Strn", "The
            Strain in each element", GH_ParamAccess.list);
    }

    protected override void SolveInstance(IGH_DataAccess DA)
    {
        //Expected inputs
        List<Line> geometry = new List<Line>();          //initial
            Geometry of lines
        double E = 0;                                    //Material
            property, initial value 210000 [MPa]
        double A = 0;                                    //Area for
            each element in same order as geometry, initial value
            10000 [mm^2]
        List<string> bdctxt = new List<string>();        //Boundary
            conditions in string format
        List<string> loadtxt = new List<string>();       //loads in
            string format


        //Set expected inputs from Indata
        if (!DA.GetDataList(0, geometry)) return;        //sets
```

```
        geometry
    if (!DA.GetDataList(1, bdctxt)) return;          //sets
        boundary conditions
    if (!DA.GetData(2, ref A)) return;               //sets Area
    if (!DA.GetData(3, ref E)) return;               //sets
        material
    if (!DA.GetDataList(4, loadtxt)) return;         //sets load


    //List all nodes (every node only once), numbering them
        according to list index
    List<Point3d> points = CreatePointList(geometry);


    //Interpret the BDC inputs (text) and create list of boundary
        condition (1/0 = free/clamped) for each dof.
    Vector<double> bdc_value = CreateBDCList(bdctxt, points);


    //Interpreting input load (text) and creating load list
        (double)
    List<double> load = CreateLoadList(loadtxt, points);


    //Create global stiffness matrix
    Matrix<double> K_tot = CreateGlobalStiffnessMatrix(geometry,
        points, E, A);


    Matrix<double> K_red;
    Vector<double> load_red;
    //Create reduced K-matrix and reduced load list (removed free
        dofs)
    CreateReducedGlobalStiffnessMatrix(bdc_value, K_tot, load,
        out K_red, out load_red);


    //Calculate deformations
    Vector<double> def_reduced = K_red.Cholesky().Solve(load_red);


    //Add the clamped dofs (= 0) to the deformations list
    Vector<double> def_tot =
        RestoreTotalDeformationVector(def_reduced, bdc_value);
```

```
            //Calculate the reaction forces from the deformations
            Vector<double> reactions = K_tot.Multiply(def_tot);
            reactions.CoerceZero(1e-8);


            List<double> internalStresses;
            List<double> internalStrains;
            //Calculate the internal strains and stresses in each member
            CalculateInternalStrainsAndStresses(def_tot, points, E, geometry, out internalStresses, out internalStrains);


            DA.SetDataList(0, def_tot);
            DA.SetDataList(1, reactions);
            DA.SetDataList(2, internalStresses);
            DA.SetDataList(3, internalStrains);
        } //End of main program

        private void CalculateInternalStrainsAndStresses(Vector<double> def, List<Point3d> points, double E, List<Line> geometry, out List<double> internalStresses, out List<double> internalStrains)
        {
            //preallocating lists
            internalStresses = new List<double>(geometry.Count);
            internalStrains = new List<double>(geometry.Count);

            foreach (Line line in geometry)
            {
                int index1 = points.IndexOf(new Point3d(Math.Round(line.From.X, 5), Math.Round(line.From.Y, 5), Math.Round(line.From.Z, 5)));
                int index2 = points.IndexOf(new Point3d(Math.Round(line.To.X, 5), Math.Round(line.To.Y, 5), Math.Round(line.To.Z, 5)));

                //fetching deformation of point
                double x1 = def[index1 * 3 + 0];
                double y1 = def[index1 * 3 + 1];
                double z1 = def[index1 * 3 + 2];
                double x2 = def[index2 * 3 + 0];
```

```
                double y2 = def[index2 * 3 + 1];
                double z2 = def[index2 * 3 + 2];

                //new node coordinates for deformed nodes
                double nx1 = points[index1].X + x1;
                double ny1 = points[index1].Y + y1;
                double nz1 = points[index1].Z + z1;
                double nx2 = points[index2].X + x2;
                double ny2 = points[index2].Y + y2;
                double nz2 = points[index2].Z + z2;

                //calculating dL = length of deformed line - original length of line
                double dL = Math.Sqrt(Math.Pow((nx2 - nx1), 2) + Math.Pow((ny2 - ny1), 2) + Math.Pow((nz2 - nz1), 2)) - line.Length;

                //calculating strain and stress
                internalStrains.Add(dL / line.Length);
                internalStresses.Add(internalStrains[internalStrains.Count - 1] * E);
            }
        }

        private Vector<double> RestoreTotalDeformationVector(Vector<double> deformations_red, Vector<double> bdc_value)
        {
            Vector<double> def = Vector<double>.Build.Dense(bdc_value.Count);
            for (int i = 0, j = 0; i < bdc_value.Count; i++)
            {
                if (bdc_value[i] == 1)
                {
                    def[i] = deformations_red[j];
                    j++;
                }
            }
            return def;
        }

        private static void CreateReducedGlobalStiffnessMatrix(Vector<double> bdc_value, Matrix<double> K, List<double> load, out Matrix<double>
```

```
            K_red, out Vector<double> load_red)
        {
            K_red = Matrix<double>.Build.SparseOfMatrix(K);
            List<double> load_redu = new List<double>(load);
            for (int i = 0, j = 0; i < load.Count; i++)
            {
                if (bdc_value[i] == 0)
                {
                    K_red = K_red.RemoveRow(i - j);
                    K_red = K_red.RemoveColumn(i - j);
                    load_redu.RemoveAt(i - j);
                    j++;
                }
            }
            load_red = Vector<double>.Build.DenseOfEnumerable(load_redu);
        }

        private Matrix<double> CreateGlobalStiffnessMatrix(List<Line> geometry, List<Point3d> points, double E, double A)
        {
            int gdofs = points.Count * 3;
            Matrix<double> KG = SparseMatrix.OfArray(new double[gdofs, gdofs]);

            foreach (Line currentLine in geometry)
            {
                double lineLength = Math.Round(currentLine.Length, 6);
                double mat = (E * A) / (lineLength);     //material properties
                Point3d p1 = new Point3d(Math.Round(currentLine.From.X, 5), Math.Round(currentLine.From.Y, 5), Math.Round(currentLine.From.Z, 5));
                Point3d p2 = new Point3d(Math.Round(currentLine.To.X, 5), Math.Round(currentLine.To.Y, 5), Math.Round(currentLine.To.Z, 5));

                double cx = (p2.X - p1.X) / lineLength;
                double cy = (p2.Y - p1.Y) / lineLength;
                double cz = (p2.Z - p1.Z) / lineLength;

                Matrix<double> T = SparseMatrix.OfArray(new double[,]
                {
                    { (cx),  (cy),  (cz), 0,0,0},
                    { (cx),  (cy),  (cz), 0,0,0},
```

```
                    { (cx), (cy), (cz), 0,0,0},
                    {0,0,0, (cx), (cy), (cz)},
                    {0,0,0, (cx), (cy), (cz)},
                    {0,0,0, (cx), (cy), (cz)},
                });

                Matrix<double> ke = DenseMatrix.OfArray(new double[,]
                                {
                    {   1,  0,  0, -1,   0,   0},
                    {   0,  0,  0,  0,   0,   0},
                    {   0,  0,  0,  0,   0,   0},
                    {   -1,  0,  0,  1,   0,   0},
                    {   0,  0,  0,  0,   0,   0},
                    {   0,  0,  0,  0,   0,   0,}
                });

                Matrix<double> T_T = T.Transpose();
                Matrix<double> Ke = ke.Multiply(T);
                Ke = T_T.Multiply(Ke);
                Ke = mat * Ke;

                int node1 = points.IndexOf(p1);
                int node2 = points.IndexOf(p2);

                //Inputting values to correct entries in Global Stiffness
                    Matrix
                for (int i = 0; i < Ke.RowCount / 2; i++)
                {

                    for (int j = 0; j < Ke.ColumnCount / 2; j++)
                    {
                        //top left 3x3 of k-element matrix
                        KG[node1 * 3 + i, node1 * 3 + j] += Ke[i, j];
                        //top right 3x3 of k-element matrix
                        KG[node1 * 3 + i, node2 * 3 + j] += Ke[i, j + 3];
                        //bottom left 3x3 of k-element matrix
                        KG[node2 * 3 + i, node1 * 3 + j] += Ke[i + 3, j];
                        //bottom right 3x3 of k-element matrix
                        KG[node2 * 3 + i, node2 * 3 + j] += Ke[i + 3, j +
                            3];
                    }
                }
            }
            return KG;
```

```
        }

        private List<double> CreateLoadList(List<string> loadtxt,
            List<Point3d> points)
        {
            List<double> loads = new List<double>(new double[points.Count
                * 3]);
            List<double> inputLoads = new List<double>();
            List<Point3d> coordlist = new List<Point3d>();

            for (int i = 0; i < loadtxt.Count; i++)
            {
                string coordstr = (loadtxt[i].Split(':')[0]);
                string loadstr = (loadtxt[i].Split(':')[1]);

                string[] coordstr1 = (coordstr.Split(','));
                string[] loadstr1 = (loadstr.Split(','));

                inputLoads.Add(Math.Round(double.Parse(loadstr1[0]), 5));
                inputLoads.Add(Math.Round(double.Parse(loadstr1[1]), 5));
                inputLoads.Add(Math.Round(double.Parse(loadstr1[2]), 5));

                coordlist.Add(new
                    Point3d(Math.Round(double.Parse(coordstr1[0]), 5),
                    Math.Round(double.Parse(coordstr1[1]), 5),
                    Math.Round(double.Parse(coordstr1[2]), 5)));
            }

            foreach (Point3d point in coordlist)
            {
                int i = points.IndexOf(point);
                int j = coordlist.IndexOf(point);
                loads[i * 3 + 0] = inputLoads[j * 3 + 0];
                loads[i * 3 + 1] = inputLoads[j * 3 + 1];
                loads[i * 3 + 2] = inputLoads[j * 3 + 2];
            }
            return loads;
        }

        private Vector<double> CreateBDCList(List<string> bdctxt,
            List<Point3d> points)
        {
            Vector<double> bdc_value =
                Vector<double>.Build.Dense(points.Count * 3, 1);
```

```
            List<int> bdcs = new List<int>();
            List<Point3d> bdc_points = new List<Point3d>(); //Coordinates relating til bdc_value in for (eg. x y z)

            for (int i = 0; i < bdctxt.Count; i++)
            {
                string coordstr = (bdctxt[i].Split(':')[0]);
                string bdcstr = (bdctxt[i].Split(':')[1]);

                string[] coordstr1 = (coordstr.Split(','));
                string[] bdcstr1 = (bdcstr.Split(','));

                bdc_points.Add(new Point3d(Math.Round(double.Parse(coordstr1[0]), 5), Math.Round(double.Parse(coordstr1[1]), 5), Math.Round(double.Parse(coordstr1[2]), 5)));

                bdcs.Add(int.Parse(bdcstr1[0]));
                bdcs.Add(int.Parse(bdcstr1[1]));
                bdcs.Add(int.Parse(bdcstr1[2]));
            }

            foreach (var point in bdc_points)
            {
                int i = points.IndexOf(point);
                bdc_value[i * 3 + 0] = bdcs[bdc_points.IndexOf(point) * 3 + 0];
                bdc_value[i * 3 + 1] = bdcs[bdc_points.IndexOf(point) * 3 + 1];
                bdc_value[i * 3 + 2] = bdcs[bdc_points.IndexOf(point) * 3 + 2];
            }
            return bdc_value;
        }

        private List<Point3d> CreatePointList(List<Line> geometry)
        {
            List<Point3d> points = new List<Point3d>();
            foreach (Line line in geometry) //adds point unless it already exists in pointlist
            {
                Point3d tempFrom = new Point3d(Math.Round(line.From.X, 5), Math.Round(line.From.Y, 5), Math.Round(line.From.Z, 5));
```

```
                    Point3d tempTo = new Point3d(Math.Round(line.To.X, 5), Math.Round(line.To.Y, 5), Math.Round(line.To.Z, 5));

                    if (!points.Contains(tempFrom))
                    {
                        points.Add(tempFrom);
                    }
                    if (!points.Contains(tempTo))
                    {
                        points.Add(tempTo);
                    }
                }
                return points;
            }

            protected override System.Drawing.Bitmap Icon
            {
                get
                {
                    return Properties.Resources.Calc;
                }
            }

            public override Guid ComponentGuid
            {
                get { return new Guid("b4e6e6ea-86b2-46dd-8475-dfa04892a212"); }
            }
        }
}
```

3D Truss Set Loads Component

```
using System;
using System.Collections.Generic;
using Grasshopper.Kernel;
using Rhino.Geometry;

namespace Truss3D
{
    public class SetLoads : GH_Component
    {
        public SetLoads()
          : base("SetLoads", "Nickname",
              "Description",
              "Koala", "Truss3D")
        {
        }

        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply load(s)", GH_ParamAccess.list);
            pManager.AddNumberParameter("Load", "L", "Load originally given i Newtons (N), give one load for all points or list of loads for each point", GH_ParamAccess.list);
            pManager.AddNumberParameter("angle (xz)", "axz", "give angle for load in xz plane", GH_ParamAccess.list, 90);
            pManager.AddNumberParameter("angle (xy)", "axy", "give angle for load in xy plane", GH_ParamAccess.list, 0);
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddTextParameter("PointLoads", "PL", "PointLoads formatted for Truss Calculation", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            //Expected inputs and output
```

```
        List<Point3d> pointList = new List<Point3d>(); //List of points where load will be applied
        List<double> loadList = new List<double>(); //List or value of load applied
        List<double> anglexz = new List<double>(); //Initial xz angle 90, angle from x axis in xz plane for load
        List<double> anglexy = new List<double>(); //Initial xy angle 0, angle from x axis in xy plane for load
        List<string> pointInStringFormat = new List<string>(); //preallocate final string output

        //Set expected inputs from Indata
        if (!DA.GetDataList(0, pointList)) return;
        if (!DA.GetDataList(1, loadList)) return;
        DA.GetDataList(2, anglexz);
        DA.GetDataList(3, anglexy);

        //initialize temporary stringline and load vectors
        string vectorString;
        double load = 0;
        double xvec = 0;
        double yvec = 0;
        double zvec = 0;

        if (loadList.Count == 1 && anglexz.Count == 1) //loads and angles are identical for all points
        {
            load = -1 * loadList[0]; //negativ load for z-dir
            xvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI / 180) * Math.Cos(anglexy[0] * Math.PI / 180), 2);
            yvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI / 180) * Math.Sin(anglexy[0] * Math.PI / 180), 2);
            zvec = Math.Round(load * Math.Sin(anglexz[0] * Math.PI / 180), 2);

            vectorString = xvec + "," + yvec + "," + zvec;
            for (int i = 0; i < pointList.Count; i++) //adds identical load to all points in pointList
            {
                pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" +
```

```
                        vectorString);
                }
            }
            else    //loads and angles may be different => calculate new xvec, yvec, zvec for all loads
            {
                for (int i = 0; i < pointList.Count; i++)
                {
                    if (loadList.Count < i)              //if pointlist is larger than loadlist, set last load value in remaining points
                    {
                        vectorString = xvec + "," + yvec + "," + zvec;
                    }
                    else
                    {
                        load = -1 * loadList[i];          //negative load for z-dir

                        xvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Cos(anglexy[i]), 2);
                        yvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Sin(anglexy[i]), 2);
                        zvec = Math.Round(load * Math.Sin(anglexz[i]), 2);

                        vectorString = xvec + "," + yvec + "," + zvec;
                    }

                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + vectorString);
                }
            }

            //Set output data
            DA.SetDataList(0, pointInStringFormat);
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.Loads;
            }
```

```
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("026f6903-826a-4012-9c39-2b18f883ba00"); }
        }
    }
}
```

3D Truss BDC Component

```
using System;
using System.Collections.Generic;
using Grasshopper.Kernel;
using Rhino.Geometry;

namespace Truss3D
{
    public class BDCComponents : GH_Component
    {
        public BDCComponents()
          : base("BDC Truss", "BDC Truss",
              "Set boundary conditions for the Truss 3D calculation",
              "Koala", "Truss3D")
        {
        }

        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply Boundary Conditions", GH_ParamAccess.list);
            pManager.AddLineParameter("Geometry", "G", "Geometry", GH_ParamAccess.list);
            pManager.AddIntegerParameter("Boundary Conditions", "BDC", "Boundary Conditions x,y,z where 0=clamped and 1=free", GH_ParamAccess.list, new List<int>(new int[] { 0, 0, 0 }));
            pManager.AddTextParameter("Locked direction", "Ldir", "Lock x, y or z direction for deformation", GH_ParamAccess.item, "");
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddTextParameter("B.Cond.", "BDC", "Boundary Conditions for 2D Truss Calculation", GH_ParamAccess.list);
        }

```

```
        protected override void SolveInstance(IGH_DataAccess DA)
        {
            //Expected inputs
            List<Point3d> pointList = new List<Point3d>();     //List of points where BDC is to be applied
            List<Line> geometry = new List<Line>();
            List<int> BDC = new List<int>();                              //is BDC free? (=clamped) (1 == true, 0 == false)
            List<string> pointInStringFormat = new List<string>();     //output in form of list of strings
            string lock_dir = "";


            //Set expected inputs from Indata and aborts with error message if input is incorrect
            if (!DA.GetDataList(0, pointList)) return;
            if (!DA.GetDataList(1, geometry)) return;
            if (!DA.GetDataList(2, BDC)) { AddRuntimeMessage(GH_RuntimeMessageLevel.Warning, "testing"); return; }
            if (!DA.GetData(3, ref lock_dir)) return;

            //Preallocate temporary variables
            string BDCString;
            int bdcx = 0;
            int bdcy = 0;
            int bdcz = 0;

            if (lock_dir == "")
            {
                if (BDC.Count == 1) //Boundary condition input for identical conditions in all points. Split into if/else for optimization
                {
                    bdcx = BDC[0];
                    bdcy = BDC[0];
                    bdcz = BDC[0];

                    BDCString = bdcx + "," + bdcy + "," + bdcz;

                    for (int i = 0; i < pointList.Count; i++)    //Format stringline for all points (identical boundary conditions for all points)
                    {
```

```
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);
                }
            }
            else if (BDC.Count == 3) //Boundary condition input for identical conditions in all points. Split into if/else for optimization
            {
                bdcx = BDC[0];
                bdcy = BDC[1];
                bdcz = BDC[2];

                BDCString = bdcx + "," + bdcy + "," + bdcz;

                for (int i = 0; i < pointList.Count; i++)   //Format stringline for all points (identical boundary conditions for all points)
                {
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);
                }
            }
            else    //BDCs are not identical for all points
            {
                for (int i = 0; i < pointList.Count; i++)
                {
                    if (i > (BDC.Count / 3) - 1)  //Are there more points than BDCs given? (BDC always lists x,y,z per point)
                    {
                        BDCString = bdcx + "," + bdcy + "," + bdcz; //use values from last BDC in list of BDCs
                    }
                    else
                    {
                        //retrieve BDC for x,y,z-dir
                        bdcx = BDC[i * 3];
                        bdcy = BDC[i * 3 + 1];
                        bdcz = BDC[i * 3 + 2];
                        BDCString = bdcx + "," + bdcy + "," + bdcz;
                    }
                    pointInStringFormat.Add(pointList[i].X + "," +
```

```
                            pointList[i].Y + "," + pointList[i].Z + ":" +
                            BDCString);    //Add stringline to list of
                            strings
                    }
                }
            }
            else
            {
                bool lx = false;
                bool ly = false;
                bool lz = false;

                if (lock_dir == "X" || lock_dir == "x")
                {
                    lx = true;
                    bdcx = 0;
                }
                else if (lock_dir == "Y" || lock_dir == "y")
                {
                    ly = true;
                    bdcy = 0;
                }
                else if (lock_dir == "Z" || lock_dir == "z")
                {
                    ly = true;
                    bdcz = 0;
                }

                List<Point3d> points = CreatePointList(geometry);
                for (int i = 0; i < pointList.Count; i++)
                {
                    points.Remove(pointList[i]);
                }
                for (int i = 0; i < points.Count; i++)
                {
                    if (!lx) bdcx = 1;
                    if (!ly) bdcy = 1;
                    if (!lz) bdcz = 1;

                    BDCString = bdcx + "," + bdcy + "," + bdcz;
                    pointInStringFormat.Add(points[i].X + "," +
                        points[i].Y + "," + points[i].Z + ":" +
                        BDCString);
                }
```

```

            if (BDC.Count == 1) //Boundary condition input for identical conditions in all points. Split into if/else for optimization
            {
                if (!lx) bdcx = BDC[0];
                if (!ly) bdcy = BDC[0];
                if (!lz) bdcz = BDC[0];

                BDCString = bdcx + "," + bdcy + "," + bdcz;

                for (int i = 0; i < pointList.Count; i++)   //Format stringline for all points (identical boundary conditions for all points)
                {
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);
                }
            }
            else if (BDC.Count == 3) //Boundary condition input for identical conditions in all points. Split into if/else for optimization
            {
                if (!lx) bdcx = BDC[0];
                if (!ly) bdcy = BDC[1];
                if (!lz) bdcz = BDC[2];

                BDCString = bdcx + "," + bdcy + "," + bdcz;

                for (int i = 0; i < pointList.Count; i++)   //Format stringline for all points (identical boundary conditions for all points)
                {
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);
                }
            }
            else    //BDCs are not identical for all points
            {
                for (int i = 0; i < pointList.Count; i++)
                {
                    if (i > (BDC.Count / 3) - 1)  //Are there more
```

```
                        points than BDCs given? (BDC always lists x,y,z per point)
                    {
                        BDCString = bdcx + "," + bdcy + "," + bdcz; //use values from last BDC in list of BDCs
                    }
                    else
                    {
                        //retrieve BDC for x,y,z-dir
                        if (!lx) bdcx = BDC[i * 3];
                        if (!ly) bdcy = BDC[i * 3 + 1];
                        if (!lz) bdcz = BDC[i * 3 + 2];
                        BDCString = bdcx + "," + bdcy + "," + bdcz;
                    }
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);    //Add stringline to list of strings
                }
            }
        }

        DA.SetDataList(0, pointInStringFormat);
    } //End of main program

    private List<Point3d> CreatePointList(List<Line> geometry)
    {
        List<Point3d> points = new List<Point3d>();

        for (int i = 0; i < geometry.Count; i++) //adds every point unless it already exists in list
        {
            Line l1 = geometry[i];
            if (!points.Contains(l1.From))
            {
                points.Add(l1.From);
            }
            if (!points.Contains(l1.To))
            {
                points.Add(l1.To);
            }
        }

        return points;
```

```
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.BDC; //Setting component icon
            }
        }

        public override Guid ComponentGuid
        {
            get { return new Guid("1376de2c-8393-45c9-81c8-512c87f6061f"); }
        }
    }
}
```

3D Truss calculation Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;

namespace Truss3D
{
    public class DeformedGeometry : GH_Component
    {
        public DeformedGeometry()
          : base("Deformed Truss", "Def.Truss",
            "Description",
            "Koala", "Truss3D")
        {
        }

        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddNumberParameter("Deformation", "Def", "The Node
                Deformation from 2DTrussCalc", GH_ParamAccess.list);
            pManager.AddLineParameter("Geometry", "G", "Input Geometry
                (Line format)", GH_ParamAccess.list);
            pManager.AddNumberParameter("Scale", "S", "The Scale Factor
                for Deformation", GH_ParamAccess.item, 1);
        }

        protected override void
            RegisterOutputParams(GH_Component.GH_OutputParamManager
            pManager)
        {
            pManager.AddLineParameter("Deformed Geometry", "Def.G.",
                "Deformed Geometry as List of Lines",
                GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            //Expected inputs and outputs
            List<double> def = new List<double>();
```

```
            List<Line> geometry = new List<Line>();
            double scale = 1;
            List<Line> defGeometry = new List<Line>();
            List<Point3d> defPoints = new List<Point3d>();

            //Set expected inputs from Indata
            if (!DA.GetDataList(0, def)) return;
            if (!DA.GetDataList(1, geometry)) return;
            if (!DA.GetData(2, ref scale)) return;

            //List all nodes (every node only once), numbering them
                according to list index
            List<Point3d> points = CreatePointList(geometry);

            int index = 0;
            //loops through all points and scales x-, y- and z-dir
            foreach (Point3d point in points)
            {
                //fetch global x,y,z placement of point
                double x = point.X;
                double y = point.Y;
                double z = point.Z;

                //scales x and z according to input Scale
                defPoints.Add(new Point3d(x + scale * def[index], y +
                    scale * def[index + 1], z + scale * def[index + 2]));
                index += 3;
            }

            //creates deformed geometry based on initial geometry
                placement
            foreach (Line line in geometry)
            {
                //fetches index of original start and endpoint
                int i1 = points.IndexOf(line.From);
                int i2 = points.IndexOf(line.To);

                //creates new line based on scaled deformation of said
                    points
                defGeometry.Add(new Line(defPoints[i1], defPoints[i2]));
            }


            //Set output data
```

```
            DA.SetDataList(0, defGeometry);
        }   //End of main program

        private List<Point3d> CreatePointList(List<Line> geometry)
        {
            List<Point3d> points = new List<Point3d>();

            for (int i = 0; i < geometry.Count; i++) //adds every point
                unless it already exists in list
            {
                Line l1 = geometry[i];
                if (!points.Contains(l1.From))
                {
                    points.Add(l1.From);
                }
                if (!points.Contains(l1.To))
                {
                    points.Add(l1.To);
                }
            }

            return points;
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.Draw;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("754421e3-67ef-49bc-b98c-354a607b163e"); }
        }
    }
}
```

Appendix C

3D Beam

3D Beam Calculation Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;
using System.Drawing;
using Grasshopper.GUI.Canvas;
using System.Windows.Forms;
using Grasshopper.GUI;

using MathNet.Numerics.LinearAlgebra;
using MathNet.Numerics.LinearAlgebra.Double;

namespace Beam3D
{
    public class CalcComponent : GH_Component
    {
        public CalcComponent()
          : base("BeamCalculation", "BeamC",
              "Description",
              "Koala", "3D Beam")
        {
        }

        //Initialize moments
```

```
        static bool startCalc = false;

        //Method to allow c hanging of variables via GUI (see Component Visual)
        public static void setStart(string s, bool i)
        {
            if (s == "Run")
            {
                startCalc = i;
            }
        }

        public override void CreateAttributes()
        {
            m_attributes = new Attributes_Custom(this);
        }

        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddLineParameter("Lines", "LNS", "Geometry, in form of Lines)", GH_ParamAccess.list);
            pManager.AddTextParameter("Boundary Conditions", "BDC", "Boundary Conditions in form x,y,z,vx,vy,vz,rx,ry,rz", GH_ParamAccess.list);
            pManager.AddTextParameter("Material properties", "Mat", "Material Properties: E, A, Iy, Iz, v, alpha (rotation about x)", GH_ParamAccess.item, "200000,3600,4920000,4920000, 0.3, 0");
            pManager.AddTextParameter("PointLoads", "PL", "Load given as Vector [N]", GH_ParamAccess.list);
            pManager.AddTextParameter("PointMoment", "PM", "Moment set in a point in [Nm]", GH_ParamAccess.list, "");
            pManager.AddIntegerParameter("Sub-Elements", "n", "Number of sub-elements", GH_ParamAccess.item, 1);
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddNumberParameter("Deformations", "Def", "Tree of Deformations", GH_ParamAccess.list);
```

```
            pManager.AddNumberParameter("Reaction Forces", "R", "Reaction Forces", GH_ParamAccess.list);
            pManager.AddNumberParameter("Applied Loads", "A", "Applied Loads", GH_ParamAccess.list);
            pManager.AddNumberParameter("Element stresses", "Strs", "The Stress in each element", GH_ParamAccess.list);
            pManager.AddNumberParameter("Element strains", "Strn", "The Strain in each element", GH_ParamAccess.list);
            pManager.AddGenericParameter("Matrix Deformations", "DM", "Deformation Matrix for def. component", GH_ParamAccess.item);
            pManager.AddPointParameter("New Base Points", "NBP", "Nodal points of sub elements", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
           #region Fetch input
            //Expected inputs
            List<Line> geometry = new List<Line>();          //Initial Geometry of lines
            List<string> bdctxt = new List<string>();        //Boundary conditions in string format
            List<string> loadtxt = new List<string>();       //loads in string format
            List<string> momenttxt = new List<string>();     //Moments in string format
            string mattxt = "";
            int n = 1;


            //Set expected inputs from Indata
            if (!DA.GetDataList(0, geometry)) return;        //sets geometry
            if (!DA.GetDataList(1, bdctxt)) return;          //sets boundary conditions as string
            if (!DA.GetData(2, ref mattxt)) return;          //sets material properties as string
            if (!DA.GetDataList(3, loadtxt)) return;         //sets load as string
            if (!DA.GetDataList(4, momenttxt)) return;       //sets moment as string
            if (!DA.GetData(5, ref n)) return;               //sets number of elements
```

```
            #endregion

            //Interpret and set material parameters
            double E;       //Material Young's modulus, initial value 200 000 [MPa]
            double A;       //Area for each element in same order as geometry, initial value CFS100x100 3600 [mm^2]
            double Iy;      //Moment of inertia about local y axis, initial value 4.92E6 [mm^4]
            double Iz;      //Moment of inertia about local z axis, initial value 4.92E6 [mm^4]
            double J;       //Polar moment of inertia
            double G;       //Shear modulus, initial value 79300 [mm^4]
            double v;       //Poisson's ratio, initial value 0.3
            double alpha;


            SetMaterial(mattxt, out E, out A, out Iy, out Iz, out J, out G, out v, out alpha);

            #region Prepares geometry, boundary conditions and loads for calculation
            //List all nodes (every node only once), numbering them according to list index
            List<Point3d> points = CreatePointList(geometry);


            //Interpret the BDC inputs (text) and create list of boundary condition (1/0 = free/clamped) for each dof.
            Vector<double> bdc_value = CreateBDCList(bdctxt, points);


            //Interpreting input load (text) and creating load list (double)
            Vector<double> load = CreateLoadList(loadtxt, momenttxt, points);
            #endregion

            Matrix<double> def_shape, glob_strain, glob_stress;
            Vector<double> reactions;
            List<Point3d> oldXYZ;

            List<Curve> defGeometry = new List<Curve>();    //output deformed geometry
```

```


        if (startCalc)
        {
            #region Create global and reduced stiffness matrix
            //Create global stiffness matrix
            Matrix<double> K_tot = GlobalStiffnessMatrix(geometry, points, E, A, Iy, Iz, J, G, alpha);


            //Create reduced K-matrix and reduced load list (removed free dofs)
            Matrix<double> KGr;
            Vector<double> load_red;
            ReducedGlobalStiffnessMatrix(bdc_value, K_tot, load, out KGr, out load_red);
            #endregion


            #region Calculate deformations, reaction forces and internal strains and stresses
            //Calculate deformations
            Vector<double> def_red = KGr.Cholesky().Solve(load_red);


            //Add the clamped dofs (= 0) to the deformations list
            Vector<double> def_tot = RestoreTotalDeformationVector(def_red, bdc_value);


            //Calculate the reaction forces from the deformations
            reactions = K_tot.Multiply(def_tot);
            reactions -= load; //method for separating reactions and applied loads
            reactions.CoerceZero(1e-8); //removing values smaller than 1e-8 arisen from numerical errors


            //Interpolate deformations using shape functions
            double y = 50;

            var z = y;
            InterpolateDeformations(def_tot, points, geometry, n, z, y, alpha, out def_shape, out oldXYZ, out glob_strain);
```

```


                //Calculate stresses
                glob_stress = E * glob_strain;
                #endregion

            }
            else
            {
                #region Set outputs to zero
                reactions = Vector<double>.Build.Dense(points.Count * 6);
                def_shape = Matrix<double>.Build.Dense(geometry.Count, 6 * (n + 1));
                glob_strain = def_shape;
                glob_stress = def_shape;

                oldXYZ = new List<Point3d>();
                #endregion
            }

            #region Format output
            double[] def = new double[def_shape.RowCount * def_shape.ColumnCount];
            for (int i = 0; i < def_shape.RowCount; i++)
            {
                for (int j = 0; j < def_shape.ColumnCount; j++)
                {
                    def[i* def_shape.ColumnCount + j] = def_shape[i, j];
                }
            }

            double[] strain = new double[glob_strain.RowCount * glob_strain.ColumnCount];
            double[] stress = new double[glob_stress.RowCount * glob_stress.ColumnCount];
            for (int i = 0; i < glob_stress.RowCount; i++)
            {
                for (int j = 0; j < glob_stress.ColumnCount; j++)
                {
                    stress[i * glob_stress.ColumnCount + j] = glob_stress[i, j];
                    strain[i * glob_stress.ColumnCount + j] = glob_strain[i, j];
                }
```

```
            }
            #endregion

            DA.SetDataList(0, def);
            DA.SetDataList(1, reactions);
            DA.SetDataList(2, load);
            DA.SetDataList(3, stress);
            DA.SetDataList(4, strain);
            DA.SetData(5, def_shape);
            DA.SetDataList(6, oldXYZ);

        } //End of main component

        private void InterpolateDeformations(Vector<double> def,
            List<Point3d> points, List<Line> geometry, int n, double
            height, double width, double alpha, out Matrix<double>
            def_shape, out List<Point3d> oldXYZ, out Matrix<double>
            glob_strain)
        {
            def_shape = Matrix<double>.Build.Dense(geometry.Count, (n +
                1) * 6);
            glob_strain = Matrix<double>.Build.Dense(geometry.Count, (n +
                1) * 3);
            Matrix<double> N, dN;
            Vector<double> u = Vector<double>.Build.Dense(12);
            oldXYZ = new List<Point3d>();
            for (int i = 0; i < geometry.Count; i++)
            {
                //fetches index of original start and endpoint
                Point3d p1 = new Point3d(Math.Round(geometry[i].From.X,
                    4), Math.Round(geometry[i].From.Y, 4),
                    Math.Round(geometry[i].From.Z, 4));
                Point3d p2 = new Point3d(Math.Round(geometry[i].To.X, 4),
                    Math.Round(geometry[i].To.Y, 4),
                    Math.Round(geometry[i].To.Z, 4));
                int i1 = points.IndexOf(p1);
                int i2 = points.IndexOf(p2);
                //create 12x1 deformation vector for element (6dofs),
                    scaled and populated with existing deformations
                for (int j = 0; j < 6; j++)
                {
                    u[j] = def[i1 * 6 + j];
                    u[j + 6] = def[i2 * 6 + j];
                }
```

```
            //interpolate points between startNode and endNode of undeformed (main) element
            List<Point3d> tempOld = InterpolatePoints(geometry[i], n);

            double L = points[i1].DistanceTo(points[i2]);   //L is distance from startnode to endnode

            //Calculate 6 dofs for all new elements using shape functions (n+1 elements)
            Matrix<double> disp = Matrix<double>.Build.Dense(n + 1, 4);
            Matrix<double> rot = Matrix<double>.Build.Dense(n + 1, 4);

            //to show scaled deformations
            Matrix<double> scaled_disp = Matrix<double>.Build.Dense(n + 1, 3);

            //transform to local coords
            var tf = TransformationMatrix(geometry[i].From, geometry[i].To, alpha);
            var T = tf.DiagonalStack(tf);
            T = T.DiagonalStack(T);
            u = T * u;

            double x = 0;
            for (int j = 0; j < n + 1; j++)
            {
                DisplacementField_NB(L, x, out N, out dN);

                disp.SetRow(j, N.Multiply(u));
                rot.SetRow(j, dN.Multiply(u));

                var d0 = new double[] { disp[j, 0], disp[j, 1], disp[j, 2] };
                var r0 = new double[] { disp[j, 3], rot[j, 2], rot[j, 1] };
                var t0 = ToGlobal(d0, r0, tf);

                disp.SetRow(j, new double[] { t0[0], t0[1], t0[2], t0[3] });
                rot.SetRow(j, new double[] { rot[j, 0], t0[5], t0[4], rot[j, 3] });
                x += L / n;
```

```
                }
                oldXYZ.AddRange(tempOld);

                //add deformation to def_shape (convert from i = nodal number to i = element number)
                def_shape.SetRow(i, SetDef(n + 1, disp, rot));

                glob_strain.SetRow(i, CalculateStrain(n, height, width, u, tf, L, def_shape)); //set strains for all subelement in current element to row i
            }
        }

        private Vector<double> CalculateStrain(int n, double height, double width, Vector<double> u, Matrix<double> tf, double L, Matrix<double> def)
        {
            Matrix<double> dN, ddN;
            double x = 0;
            var strains = Vector<double>.Build.Dense((n + 1) * 3); //contains all subelement strains (only for one element)
            for (int j = 0; j < n + 1; j++)
            {
                DisplacementField_ddN(L, x, out ddN);
                DisplacementField_dN(L, x, out dN);

                //u and N are in local coordinates
                var tmp1 = dN * u; //tmp1 = du_x, du_y, du_z, dtheta_x
                var tmp2 = ddN * u; //tmp2 = ddu_x/dx, ddu_y/dx, ddu_z_dx, ddtheta_x/dx

                strains[j * 3] = tmp1[0];
                strains[j * 3 + 1] = height * tmp2[2];
                strains[j * 3 + 2] = width * tmp2[1];

                x += L / n;
            }
            return strains;
        }

        private Vector<double> ToGlobal(double[] d, double[] r, Matrix<double> tf)
        {
            var dr = Vector<double>.Build.Dense(6);
```

```
            for (int i = 0; i < 3; i++)
            {
                dr[i] = d[i];
                dr[i + 3] = r[i];
            }
            tf = tf.DiagonalStack(tf);

            dr = tf.Transpose() * dr;
            return dr;
        }

        private double[] SetDef(int m, Matrix<double> disp, Matrix<double> rot)
        {
            //m == n+1
            double[] def_e = new double[m * 6];
            for (int i = 0; i < m; i++)
            {
                //add displacements in x,y,z
                def_e[i * 6 + 0] = disp[i, 0];
                def_e[i * 6 + 1] = disp[i, 1];
                def_e[i * 6 + 2] = disp[i, 2];
                //add rotations
                def_e[i * 6 + 3] = disp[i, 3];
                def_e[i * 6 + 4] = rot[i, 2]; //theta_y = d_uz/d_x
                def_e[i * 6 + 5] = rot[i, 1]; //theta_z = d_uy/d_x
            }
            return def_e;
        }

        private List<Point3d> InterpolatePoints(Line line, int n)
        {
            List<Point3d> tempP = new List<Point3d>(n + 1);
            double[] t = LinSpace(0, 1, n + 1);
            for (int i = 0; i < t.Length; i++)
            {
                var tPm = new Point3d();
                tPm.Interpolate(line.From, line.To, t[i]);
                tPm = new Point3d(Math.Round(tPm.X, 4), Math.Round(tPm.Y, 4), Math.Round(tPm.Z, 4));
                tempP.Add(tPm);
            }
            return tempP;
        }
```

```
        private static double[] LinSpace(double x1, double x2, int n)
        {
            //Generate a 1-D array of linearly spaced values
            double step = (x2 - x1) / (n - 1);
            double[] y = new double[n];
            for (int i = 0; i < n; i++)
            {
                y[i] = x1 + step * i;
            }
            return y;
        }

        private void DisplacementField_NB(double L, double x, out Matrix<double> N, out Matrix<double> dN)
        {
            double N1 = 1 - x / L;
            double N2 = x / L;
            double N3 = 1 - 3 * Math.Pow(x, 2) / Math.Pow(L, 2) + 2 * Math.Pow(x, 3) / Math.Pow(L, 3);
            double N4 = x - 2 * Math.Pow(x, 2) / L + Math.Pow(x, 3) / Math.Pow(L, 2);
            double N5 = -N3 + 1;//3 * Math.Pow(x, 2) / Math.Pow(L, 2) - 2 * Math.Pow(x, 3) / Math.Pow(L, 3);
            double N6 = Math.Pow(x, 3) / Math.Pow(L, 2) - Math.Pow(x, 2) / L;

            N = Matrix<double>.Build.DenseOfArray(new double[,] {
                { N1, 0, 0,  0,  0,  0, N2, 0,  0,  0,  0,  0},
                { 0, N3, 0,  0,  0, N4, 0, N5, 0,  0,  0, N6 },
                { 0, 0, N3, 0, -N4, 0, 0, 0, N5, 0, -N6, 0},
                { 0, 0, 0, N1, 0, 0, 0, 0, 0, N2, 0, 0} });

            //u = [u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, u11, u12]
            //u = [ux, uy, uz, theta_x]

            double dN1 = -1 / L;
            double dN2 = 1 / L;
            double dN3 = -6 * x / Math.Pow(L, 2) + 6 * Math.Pow(x, 2) / Math.Pow(L, 3);
            double dN4 = 3 * Math.Pow(x, 2) / Math.Pow(L, 2) - 4 * x / L + 1;
            double dN5 = -dN3;//6 * x / Math.Pow(L, 2) - 6 * Math.Pow(x, 2) / Math.Pow(L, 3);
```

```
            double dN6 = 3 * Math.Pow(x, 2) / Math.Pow(L, 2) - 2 * x / L;

            dN = Matrix<double>.Build.DenseOfArray(new double[,] {
                { dN1, 0,     0,   0,   0,      0,  dN2,     0,   0,   0,     0,
                        0},
                { 0, dN3,     0,   0,   0,  dN4,     0,  dN5,     0,   0,     0,
                    dN6 },
                { 0,     0, dN3,   0, dN4,    0,   0,       0,dN5,   0, dN6,
                    0},
                //{ 0,     0, dN3,   0, -dN4,    0,   0,       0,dN5,   0,
                    -dN6,     0},
                { 0,     0,     0, dN1, 0,      0,   0,       0,   0, dN2,   0,
                        0} });

            //theta_y = du_z/dx
            //theta_z = du_y/dx
        }

        private void DisplacementField_dN(double L, double x, out
            Matrix<double> dN)
        {
            double dN1 = -1 / L;
            double dN2 = 1 / L;
            double dN3 = -6 * x / Math.Pow(L, 2) + 6 * Math.Pow(x, 2) /
                Math.Pow(L, 3);
            double dN4 = 3 * Math.Pow(x, 2) / Math.Pow(L, 2) - 4 * x / L
                + 1;
            double dN5 = -dN3;//6 * x / Math.Pow(L, 2) - 6 * Math.Pow(x,
                2) / Math.Pow(L, 3);
            double dN6 = 3 * Math.Pow(x, 2) / Math.Pow(L, 2) - 2 * x / L;

            dN = Matrix<double>.Build.DenseOfArray(new double[,] {
                { dN1, 0,     0,   0,   0,      0,  dN2,     0,   0,   0,     0,
                        0},
                { 0, dN3,     0,   0,   0,  dN4,     0,  dN5,     0,   0,     0,
                    dN6 },
                { 0,     0, dN3,   0, dN4,    0,   0,       0,dN5,   0, dN6,
                    0},
                { 0,     0,     0, dN1, 0,      0,   0,       0,   0, dN2,   0,
                        0} });

            //theta_y = du_z/dx
            //theta_z = du_y/dx
        }
```

```
        private void DisplacementField_ddN(double L, double x, out
            Matrix<double> ddN)
        {
            double ddN1 = 0;
            double ddN2 = 0;
            double ddN3 = -6 / Math.Pow(L, 2) + 12 * x / Math.Pow(L, 3);
            double ddN4 = -4 / L + 6 * x / Math.Pow(L, 2);
            double ddN5 = 6 / Math.Pow(L, 2) - 12 * x / Math.Pow(L, 3);
            double ddN6 = 6 * x / Math.Pow(L, 2) - 2 / L;

            ddN = Matrix<double>.Build.DenseOfArray(new double[,] {
                { ddN1, 0, 0,  0,  0,  0, ddN2, 0,  0,  0,  0,  0},
                { 0, ddN3, 0,  0,  0, ddN4, 0, ddN5, 0,  0,  0, ddN6 },
                { 0, 0, ddN3, 0, -ddN4,  0, 0, 0, ddN5, 0, -ddN6, 0},
                { 0, 0, 0,  ddN1,  0,  0, 0, 0,  0, ddN2, 0, 0}
            });
        }

        private Vector<double>
            RestoreTotalDeformationVector(Vector<double>
            deformations_red, Vector<double> bdc_value)
        {
            Vector<double> def =
                Vector<double>.Build.Dense(bdc_value.Count);
            for (int i = 0, j = 0; i < bdc_value.Count; i++)
            {
                //if deformation has been calculated, it is added to the
                    vector. Otherwise, the deformation is zero.
                if (bdc_value[i] == 1)
                {
                    def[i] = deformations_red[j];
                    j++;
                }
            }
            return def;
        }

        private void ReducedGlobalStiffnessMatrix(Vector<double>
            bdc_value, Matrix<double> K, Vector<double> load, out
            Matrix<double> KGr, out Vector<double> load_red)
        {
            int oldRC = load.Count;
            int newRC = Convert.ToInt16(bdc_value.Sum());
```

```
            KGr = Matrix<double>.Build.Dense(newRC, newRC);
            load_red = Vector<double>.Build.Dense(newRC, 0);
            for (int i = 0, ii = 0; i < oldRC; i++)
            {
                //is bdc_value in row i free?
                if (bdc_value[i] == 1)
                {
                    for (int j = 0, jj = 0; j <= i; j++)
                    {
                        //is bdc_value in col j free?
                        if (bdc_value[j] == 1)
                        {
                            //if yes, then add to new K
                            KGr[i - ii, j - jj] = K[i, j];
                            KGr[j - jj, i - ii] = K[i, j];
                        }
                        else
                        {
                            //if not, remember to skip 1 column when adding next time (default matrix value is 0)
                            jj++;
                        }
                    }
                    //add load to reduced list
                    load_red[i - ii] = load[i];
                }
                else
                {
                    //if not, remember to skip 1 row when adding next time (default matrix value is 0)
                    ii++;
                }
            }
        }

        private Matrix<double> TransformationMatrix(Point3d p1, Point3d p2, double alpha)
        {
            double L = p1.DistanceTo(p2);

            double cx = (p2.X - p1.X) / L;
            double cy = (p2.Y - p1.Y) / L;
            double cz = (p2.Z - p1.Z) / L;
```

```
            double c1 = Math.Cos(alpha);
            double s1 = Math.Sin(alpha);
            double cxz = Math.Round(Math.Sqrt(Math.Pow(cx, 2) +
                Math.Pow(cz, 2)), 6);

            Matrix<double> t;

            if (Math.Round(cx, 6) == 0 && Math.Round(cz, 6) == 0)
            {
                t = Matrix<double>.Build.DenseOfArray(new double[,]
            {
                        {      0, cy,   0},
                        { -cy*c1,  0, s1},
                        {  cy*s1,  0, c1},
            });
            }
            else
            {
                t = Matrix<double>.Build.DenseOfArray(new double[,]
            {
                        {                    cx,       cy,
                            cz},
                        {(-cx*cy*c1 - cz*s1)/cxz,
                            cxz*c1,(-cy*cz*c1+cx*s1)/cxz},
                        {   (cx*cy*s1-cz*c1)/cxz,  -cxz*s1,
                            (cy*cz*s1+cx*c1)/cxz},
            });
            }
            return t;
        }

        private void ElementStiffnessMatrix(Line currentLine, double E,
            double A, double Iy, double Iz, double J, double G, double
            alpha, out Point3d p1, out Point3d p2, out Matrix<double> Ke)
        {
            double L = Math.Round(currentLine.Length, 6);

            p1 = new Point3d(Math.Round(currentLine.From.X, 4),
                Math.Round(currentLine.From.Y, 4),
                Math.Round(currentLine.From.Z, 4));
            p2 = new Point3d(Math.Round(currentLine.To.X, 4),
                Math.Round(currentLine.To.Y, 4),
                Math.Round(currentLine.To.Z, 4));

```

```
            Matrix<double> tf = TransformationMatrix(p1, p2, alpha);
            var T = tf.DiagonalStack(tf);
            T = T.DiagonalStack(T);

            Matrix<double> T_T = T.Transpose();

            double A1 = (E * A) / (L);

            double kz1 = (12 * E * Iz) / (L * L * L);
            double kz2 = (6 * E * Iz) / (L * L);
            double kz3 = (4 * E * Iz) / L;
            double kz4 = (2 * E * Iz) / L;

            double ky1 = (12 * E * Iy) / (L * L * L);
            double ky2 = (6 * E * Iy) / (L * L);
            double ky3 = (4 * E * Iy) / L;
            double ky4 = (2 * E * Iy) / L;

            double C1 = (G * J) / L;

            Matrix<double> ke = DenseMatrix.OfArray(new double[,]
            {
                    { A1,    0,    0,    0,    0,    0,  -A1,    0,    0,
                        0,    0,    0 },
                    {  0,  kz1,    0,    0,    0,  kz2,    0, -kz1,    0,
                        0,    0,  kz2 },
                    {  0,    0,  ky1,    0, -ky2,    0,    0,    0, -ky1,
                        0, -ky2,    0 },
                    {  0,    0,    0,   C1,    0,    0,    0,    0,    0,
                      -C1,    0,    0 },
                    {  0,    0, -ky2,    0,  ky3,    0,    0,    0,  ky2,
                        0,  ky4,    0 },
                    {  0,  kz2,    0,    0,    0,  kz3,    0, -kz2,    0,
                        0,    0,  kz4 },
                    {-A1,    0,    0,    0,    0,    0,   A1,    0,    0,
                        0,    0,    0 },
                    {  0, -kz1,    0,    0,    0, -kz2,    0,  kz1,    0,
                        0,    0, -kz2 },
                    {  0,    0, -ky1,    0,  ky2,    0,    0,    0,  ky1,
                        0,  ky2,    0 },
                    {  0,    0,    0,  -C1,    0,    0,    0,    0,    0,
                       C1,    0,    0 },
                    {  0,    0, -ky2,    0,  ky4,    0,    0,    0,  ky2,
                        0,  ky3,    0 },
```

```
                    {   0,   kz2,     0,     0,     0,   kz4,     0, -kz2,     0,     0,     0,   kz3 },
            });

            ke = ke.Multiply(T);
            Ke = T_T.Multiply(ke);
        }

        private Matrix<double> GlobalStiffnessMatrix(List<Line> geometry, List<Point3d> points, double E, double A, double Iy, double Iz, double J, double G, double alpha)
        {
            int gdofs = points.Count * 6;
            Matrix<double> KG = DenseMatrix.OfArray(new double[gdofs, gdofs]);

            foreach (Line currentLine in geometry)
            {
                Matrix<double> Ke;
                Point3d p1, p2;

                //Calculate Ke
                ElementStiffnessMatrix(currentLine, E, A, Iy, Iz, J, G, alpha, out p1, out p2, out Ke);

                //Fetch correct point indices
                int node1 = points.IndexOf(p1);
                int node2 = points.IndexOf(p2);

                //Inputting Ke to correct entries in Global Stiffness Matrix
                for (int i = 0; i < Ke.RowCount / 2; i++)
                {
                    for (int j = 0; j < Ke.ColumnCount / 2; j++)
                    {
                        //top left 3x3 of k-element matrix
                        KG[node1 * 6 + i, node1 * 6 + j] += Ke[i, j];
                        //top right 3x3 of k-element matrix
                        KG[node1 * 6 + i, node2 * 6 + j] += Ke[i, j + 6];
                        //bottom left 3x3 of k-element matrix
                        KG[node2 * 6 + i, node1 * 6 + j] += Ke[i + 6, j];
                        //bottom right 3x3 of k-element matrix
                        KG[node2 * 6 + i, node2 * 6 + j] += Ke[i + 6, j + 6];
```

```
                }
            }
        }
        return KG;
    }

    private Vector<double> CreateLoadList(List<string> loadtxt,
        List<string> momenttxt, List<Point3d> points)
    {
        Vector<double> loads =
            Vector<double>.Build.Dense(points.Count * 6);
        List<double> inputLoads = new List<double>();
        List<Point3d> coordlist = new List<Point3d>();

        for (int i = 0; i < loadtxt.Count; i++)
        {
            string coordstr = (loadtxt[i].Split(':')[0]);
            string loadstr = (loadtxt[i].Split(':')[1]);

            string[] coordstr1 = (coordstr.Split(','));
            string[] loadstr1 = (loadstr.Split(','));

            inputLoads.Add(Math.Round(double.Parse(loadstr1[0]), 4));
            inputLoads.Add(Math.Round(double.Parse(loadstr1[1]), 4));
            inputLoads.Add(Math.Round(double.Parse(loadstr1[2]), 4));

            coordlist.Add(new
                Point3d(Math.Round(double.Parse(coordstr1[0]), 4),
                Math.Round(double.Parse(coordstr1[1]), 4),
                Math.Round(double.Parse(coordstr1[2]), 4)));
        }

        foreach (Point3d point in coordlist)
        {
            int i = points.IndexOf(point);
            int j = coordlist.IndexOf(point);
            loads[i * 6 + 0] = inputLoads[j * 3 + 0]; //is loads out
                of range? (doesn't seem to have been initialized with
                size yet)
            loads[i * 6 + 1] = inputLoads[j * 3 + 1];
            loads[i * 6 + 2] = inputLoads[j * 3 + 2];
        }
        inputLoads.Clear();
        coordlist.Clear();
```

```
            for (int i = 0; i < momenttxt.Count; i++) if (momenttxt[0] !=
                "")
                {
                    string coordstr = (momenttxt[i].Split(':')[0]);
                    string loadstr = (momenttxt[i].Split(':')[1]);

                    string[] coordstr1 = (coordstr.Split(','));
                    string[] loadstr1 = (loadstr.Split(','));

                    inputLoads.Add(Math.Round(double.Parse(loadstr1[0]),
                        4));
                    inputLoads.Add(Math.Round(double.Parse(loadstr1[1]),
                        4));
                    inputLoads.Add(Math.Round(double.Parse(loadstr1[2]),
                        4));


                    coordlist.Add(new
                        Point3d(Math.Round(double.Parse(coordstr1[0]),
                        4), Math.Round(double.Parse(coordstr1[1]), 4),
                        Math.Round(double.Parse(coordstr1[2]), 4)));
                }

            foreach (Point3d point in coordlist)
            {
                int i = points.IndexOf(point);
                int j = coordlist.IndexOf(point);
                loads[i * 6 + 3] = inputLoads[j * 3 + 0];
                loads[i * 6 + 4] = inputLoads[j * 3 + 1];
                loads[i * 6 + 5] = inputLoads[j * 3 + 2];
            }
            return loads;
        }

        private Vector<double> CreateBDCList(List<string> bdctxt,
            List<Point3d> points)
        {
            //initializing bdc_value as vector of size gdofs, and entry
                values = 1
            Vector<double> bdc_value = Vector.Build.Dense(points.Count *
                6, 1);
            List<int> bdcs = new List<int>();
            List<Point3d> bdc_points = new List<Point3d>(); //Coordinates
                relating til bdc_value in for (eg. x y z)
```

```
            //Parse string input
            for (int i = 0; i < bdctxt.Count; i++)
            {
                string coordstr = (bdctxt[i].Split(’:’)[0]);
                string bdcstr = (bdctxt[i].Split(’:’)[1]);

                string[] coordstr1 = (coordstr.Split(’,’));
                string[] bdcstr1 = (bdcstr.Split(’,’));

                bdc_points.Add(new
                    Point3d(Math.Round(double.Parse(coordstr1[0]), 4),
                    Math.Round(double.Parse(coordstr1[1]), 4),
                    Math.Round(double.Parse(coordstr1[2]), 4)));

                bdcs.Add(int.Parse(bdcstr1[0]));
                bdcs.Add(int.Parse(bdcstr1[1]));
                bdcs.Add(int.Parse(bdcstr1[2]));
                bdcs.Add(int.Parse(bdcstr1[3]));
                bdcs.Add(int.Parse(bdcstr1[4]));
                bdcs.Add(int.Parse(bdcstr1[5]));
            }

            //Format to correct entries in bdc_value
            foreach (var point in bdc_points)
            {
                int globalI = points.IndexOf(point);
                int localI = bdc_points.IndexOf(point);
                bdc_value[globalI * 6 + 0] = bdcs[localI * 6 + 0];
                bdc_value[globalI * 6 + 1] = bdcs[localI * 6 + 1];
                bdc_value[globalI * 6 + 2] = bdcs[localI * 6 + 2];
                bdc_value[globalI * 6 + 3] = bdcs[localI * 6 + 3];
                bdc_value[globalI * 6 + 4] = bdcs[localI * 6 + 4];
                bdc_value[globalI * 6 + 5] = bdcs[localI * 6 + 5];
            }
            return bdc_value;
        }

        private void SetMaterial(string mattxt, out double E, out double
            A, out double Iy, out double Iz, out double J, out double G,
            out double v, out double alpha)
        {
            string[] matProp = (mattxt.Split(’,’));

```

```
            E = (Math.Round(double.Parse(matProp[0]), 2));
            A = (Math.Round(double.Parse(matProp[1]), 2));
            Iy = (Math.Round(double.Parse(matProp[2]), 2));
            Iz = (Math.Round(double.Parse(matProp[3]), 2));
            v = (Math.Round(double.Parse(matProp[4]), 2));
            G = E / (2 * (1 + Math.Pow(v, 2)));
            alpha = (Math.Round(double.Parse(matProp[5]),
                2))*Math.PI/180; //to radians

            J = Iy + Iz;
        }

        private List<Point3d> CreatePointList(List<Line> geometry)
        {
            List<Point3d> points = new List<Point3d>();
            foreach (Line line in geometry) //adds point unless it
                already exists in pointlist
            {
                Point3d tempFrom = new Point3d(Math.Round(line.From.X,
                    4), Math.Round(line.From.Y, 4),
                    Math.Round(line.From.Z, 4));
                Point3d tempTo = new Point3d(Math.Round(line.To.X, 4),
                    Math.Round(line.To.Y, 4), Math.Round(line.To.Z, 4));

                if (!points.Contains(tempFrom))
                {
                    points.Add(tempFrom);
                }
                if (!points.Contains(tempTo))
                {
                    points.Add(tempTo);
                }
            }
            return points;
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.Calc;
            }
        }

```

```
        public override Guid ComponentGuid
        {
            get { return new
                Guid("d636ebc9-0d19-44d5-a3ad-cec704b82323"); }
        }


        /// Component Visual//
        public class Attributes_Custom :
            Grasshopper.Kernel.Attributes.GH_ComponentAttributes
        {
            public Attributes_Custom(GH_Component owner) : base(owner) { }
            protected override void Layout()
            {
                base.Layout();

                Rectangle rec0 = GH_Convert.ToRectangle(Bounds);

                rec0.Height += 22;

                Rectangle rec1 = rec0;
                rec1.X = rec0.Left + 1;
                rec1.Y = rec0.Bottom - 22;
                rec1.Width = (rec0.Width) / 3 + 1;
                rec1.Height = 22;
                rec1.Inflate(-2, -2);

                Bounds = rec0;
                ButtonBounds = rec1;

            }

            GH_Palette xColor = GH_Palette.Grey;

            private Rectangle ButtonBounds { get; set; }

            protected override void Render(GH_Canvas canvas, Graphics
                graphics, GH_CanvasChannel channel)
            {
                base.Render(canvas, graphics, channel);
                if (channel == GH_CanvasChannel.Objects)
                {
                    GH_Capsule button =
                        GH_Capsule.CreateTextCapsule(ButtonBounds,
```

```
                        ButtonBounds, xColor, "Run", 3, 0);
                    button.Render(graphics, Selected, false, false);
                    button.Dispose();
                }
            }

            public override GH_ObjectResponse
                RespondToMouseDown(GH_Canvas sender, GH_CanvasMouseEvent
                e)
            {
                if (e.Button == MouseButtons.Left)
                {
                    RectangleF rec = ButtonBounds;
                    if (rec.Contains(e.CanvasLocation))
                    {
                        switchColor("Run");
                        if (xColor == GH_Palette.Black) {
                            CalcComponent.setStart("Run", true);
                            Owner.ExpireSolution(true); }
                        if (xColor == GH_Palette.Grey) {
                            CalcComponent.setStart("Run", false); }
                        sender.Refresh();
                        return GH_ObjectResponse.Handled;
                    }
                }
                return base.RespondToMouseDown(sender, e);
            }

            private void switchColor(string button)
            {
                if (button == "Run")
                {
                    if (xColor == GH_Palette.Black) { xColor =
                        GH_Palette.Grey; }
                    else { xColor = GH_Palette.Black; }
                }
            }
        }
    }
}
```

3D Beam SetLoads Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;

namespace Beam3D
{
    public class SetLoads : GH_Component
    {
        public SetLoads()
          : base("SetLoads", "SL",
              "Description",
              "Koala", "3D Beam")
        {
        }
        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply
                load(s)", GH_ParamAccess.list);
            pManager.AddNumberParameter("Load", "L", "Load originally
                given i Newtons (N), give one load for all points or list
                of loads for each point", GH_ParamAccess.list);
            pManager.AddNumberParameter("angle (xz)", "axz", "give angle
                for load in xz plane", GH_ParamAccess.list, 90);
            pManager.AddNumberParameter("angle (xy)", "axy", "give angle
                for load in xy plane", GH_ParamAccess.list, 0);
       }

        protected override void
            RegisterOutputParams(GH_Component.GH_OutputParamManager
            pManager)
        {
            pManager.AddTextParameter("PointLoads", "PL", "PointLoads
                formatted for Truss Calculation", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch inputs
            //Expected inputs and output
```

```
List<Point3d> pointList = new List<Point3d>(); //List of points where load will be applied
List<double> loadList = new List<double>(); //List or value of load applied
List<double> anglexz = new List<double>(); //Initial xz angle 90, angle from x axis in xz plane for load
List<double> anglexy = new List<double>(); //Initial xy angle 0, angle from x axis in xy plane for load
List<string> pointInStringFormat = new List<string>(); //preallocate final string output

//Set expected inputs from Indata
if (!DA.GetDataList(0, pointList)) return;
if (!DA.GetDataList(1, loadList)) return;
DA.GetDataList(2, anglexz);
DA.GetDataList(3, anglexy);
#endregion

#region Format pointloads
//initialize temporary stringline and load vectors
string vectorString;
double load = 0;
double xvec = 0;
double yvec = 0;
double zvec = 0;

if (loadList.Count == 1 && anglexz.Count == 1) //loads and angles are identical for all points
{
    load = -1 * loadList[0]; //negativ load for z-dir
    xvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI / 180) * Math.Cos(anglexy[0] * Math.PI / 180), 4);
    yvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI / 180) * Math.Sin(anglexy[0] * Math.PI / 180), 4);
    zvec = Math.Round(load * Math.Sin(anglexz[0] * Math.PI / 180), 4);

    vectorString = xvec + "," + yvec + "," + zvec;
    for (int i = 0; i < pointList.Count; i++) //adds identical load to all points in pointList
    {
```

```
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + vectorString);
                }
            }
            else    //loads and angles may be different => calculate new xvec, yvec, zvec for all loads
            {
                for (int i = 0; i < pointList.Count; i++)
                {
                    if (loadList.Count < i)             //if pointlist is larger than loadlist, set last load value in remaining points
                    {
                        vectorString = xvec + "," + yvec + "," + zvec;
                    }
                    else
                    {
                        load = -1 * loadList[i];        //negative load for z-dir

                        xvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Cos(anglexy[i]), 4);
                        yvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Sin(anglexy[i]), 4);
                        zvec = Math.Round(load * Math.Sin(anglexz[i]), 4);

                        vectorString = xvec + "," + yvec + "," + zvec;
                    }

                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + vectorString);
                }
            }
            #endregion

            //Set output data
            DA.SetDataList(0, pointInStringFormat);
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
```

```
            {
                return Properties.Resources.Pointloads;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("97664d05-2d53-4d61-a027-b71beebb9f48"); }
        }
    }
}
```

3D Beam SetMoments Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;
using System.Drawing;
using Grasshopper.GUI.Canvas;
using System.Windows.Forms;
using Grasshopper.GUI;

namespace Beam3D
{
    public class SetMoments : GH_Component
    {
        public SetMoments()
          : base("SetMoments", "Nickname",
                "Description",
                "Koala", "3D Beam")
        {
        }
        //Initialize moments
        static int mx;
        static int my;
        static int mz;


        //Method to allow c hanging of variables via GUI (see Component
            Visual)
```

```
        public static void setMom(string s, int i)
        {
            if (s == "MX")
            {
                mx = i;
            }
            else if (s == "MY")
            {
                my = i;
            }
            else if (s == "MZ")
            {
                mz = i;
            }
        }

        public override void CreateAttributes()
        {
            m_attributes = new Attributes_Custom(this);
        }

        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply moment", GH_ParamAccess.list);
            pManager.AddNumberParameter("Moment", "M", "Moment Magnitude [kNm]", GH_ParamAccess.list);
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddTextParameter("MomentLoads", "ML", "MomentLoads formatted for Beam Calculation", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch inputs
            //Expected inputs and output
            List<Point3d> pointList = new List<Point3d>();
```

```
            //List of points where load will be applied
        List<double> momentList = new List<double>();
            //List or value of load applied
        List<string> pointInStringFormat = new List<string>();
            //preallocate final string output

        //Set expected inputs from Indata
        if (!DA.GetDataList(0, pointList)) return;
        if (!DA.GetDataList(1, momentList)) return;
        #endregion

        #region Format output
        string vectorString;

        for (int i = 0, j = 0; i < pointList.Count; i++)    //Format
            stringline for all points (identical boundary conditions
            for all points)
        {
                vectorString = momentList[j] * mx + "," +
                    momentList[j] * my + "," + momentList[j] * mz;
                pointInStringFormat.Add(pointList[i].X + "," +
                    pointList[i].Y + "," + pointList[i].Z + ":" +
                    vectorString);
            if (j < momentList.Count - 1)
            {
                j++;
            }
        }
        #endregion

        //Set output data
        DA.SetDataList(0, pointInStringFormat);
    }

    protected override System.Drawing.Bitmap Icon
    {
        get
        {
            return Properties.Resources.Moments;
        }
    }

    public override Guid ComponentGuid
    {
```

```
            get { return new Guid("540c5cd8-b017-45d3-b3d1-cb1bf0c9051c"); }
        }

        /// Component Visual//
        public class Attributes_Custom : Grasshopper.Kernel.Attributes.GH_ComponentAttributes
        {
            public Attributes_Custom(GH_Component owner) : base(owner) { }
            protected override void Layout()
            {
                base.Layout();

                Rectangle rec0 = GH_Convert.ToRectangle(Bounds);

                rec0.Height += 22;

                Rectangle rec1 = rec0;
                rec1.X = rec0.Left + 1;
                rec1.Y = rec0.Bottom - 22;
                rec1.Width = (rec0.Width) / 3 + 1;
                rec1.Height = 22;
                rec1.Inflate(-2, -2);

                Rectangle rec2 = rec1;
                rec2.X = rec1.Right + 2;

                Rectangle rec3 = rec2;
                rec3.X = rec2.Right + 2;

                BoundsAllButtons = rec0;
                Bounds = rec0;
                ButtonBounds = rec1;
                ButtonBounds2 = rec2;
                ButtonBounds3 = rec3;
            }

            GH_Palette xColor = GH_Palette.Grey;
            GH_Palette yColor = GH_Palette.Grey;
            GH_Palette zColor = GH_Palette.Grey;

            private Rectangle BoundsAllButtons { get; set; }
            private Rectangle ButtonBounds { get; set; }
            private Rectangle ButtonBounds2 { get; set; }
```

```
        private Rectangle ButtonBounds3 { get; set; }

        protected override void Render(GH_Canvas canvas, Graphics
            graphics, GH_CanvasChannel channel)
        {
            base.Render(canvas, graphics, channel);
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button =
                    GH_Capsule.CreateTextCapsule(ButtonBounds,
                    ButtonBounds, xColor, "MX", 2, 0);
                button.Render(graphics, Selected, Owner.Locked,
                    false);
                button.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button2 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds2,
                    ButtonBounds2, yColor, "MY", 2, 0);
                button2.Render(graphics, Selected, Owner.Locked,
                    false);
                button2.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button3 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds3,
                    ButtonBounds3, zColor, "MZ", 2, 0);
                button3.Render(graphics, Selected, Owner.Locked,
                    false);
                button3.Dispose();
            }
        }

        public override GH_ObjectResponse
            RespondToMouseDown(GH_Canvas sender, GH_CanvasMouseEvent
            e)
        {
            if (e.Button == MouseButtons.Left)
            {
                RectangleF rec = ButtonBounds;
                if (rec.Contains(e.CanvasLocation))
                {
```

```
                    switchColor("MX");
                }
                rec = ButtonBounds2;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("MY");
                }
                rec = ButtonBounds3;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("MZ");
                }
                rec = BoundsAllButtons;
                if (rec.Contains(e.CanvasLocation))
                {
                    if (xColor == GH_Palette.Grey) { setMom("MX", 0); }
                    else { setMom("MX", 1); }
                    if (yColor == GH_Palette.Grey) { setMom("MY", 0); }
                    else { setMom("MY", 1); }
                    if (zColor == GH_Palette.Grey) { setMom("MZ", 0); }
                    else { setMom("MZ", 1); }
                    Owner.ExpireSolution(true);
                }
                return GH_ObjectResponse.Handled;
            }
            return base.RespondToMouseDown(sender, e);
        }

        private void switchColor(string button)
        {
            if (button == "MX")
            {
                if (xColor == GH_Palette.Black) { xColor = GH_Palette.Grey; }
                else { xColor = GH_Palette.Black; }
            }
            else if (button == "MY")
            {
                if (yColor == GH_Palette.Black) { yColor = GH_Palette.Grey; }
                else { yColor = GH_Palette.Black; }
```

```
                }
                else if (button == "MZ")
                {
                    if (zColor == GH_Palette.Black) { zColor =
                        GH_Palette.Grey; }
                    else { zColor = GH_Palette.Black; }
                }
                Owner.ExpireSolution(true);
            }
        }
    }
}
```

3D Beam BDC Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;
using System.Drawing;
using Grasshopper.GUI.Canvas;
using System.Windows.Forms;
using Grasshopper.GUI;

namespace Beam3D
{
    public class BDCComponent : GH_Component
    {
        public BDCComponent()
          : base("BDCComponent", "BDCs",
              "Description",
              "Koala", "3D Beam")
        {
        }

        //Initialize BDCs
        private static int x;
        private static int y;
        private static int z;
        private static int rx;
        private static int ry;
        private static int rz;
```

```


        //Method to allow c hanging of variables via GUI (see Component Visual)
        private static void setBDC(string s, int i)
        {
            if (s == "X")
            {
                x = i;
            }
            else if (s == "Y")
            {
                y = i;
            }
            else if (s == "Z")
            {
                z = i;
            }
            else if (s == "RX")
            {
                rx = i;
            }
            else if (s == "RY")
            {
                ry = i;
            }
            else if (s == "RZ")
            {
                rz = i;
            }
        }

        public override void CreateAttributes()
        {
            m_attributes = new Attributes_Custom(this);
        }

        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply Boundary Conditions", GH_ParamAccess.list);
        }
```

```
        protected override void
            RegisterOutputParams(GH_Component.GH_OutputParamManager
            pManager)
        {
            pManager.AddTextParameter("B.Cond.", "BDC", "Boundary
                Conditions for 3D Beam Calculation", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch inputs
            //Expected inputs
            List<Point3d> pointList = new List<Point3d>();
                //List of points where BDC is to be applied
            List<string> pointInStringFormat = new List<string>();
                //output in form of list of strings


            //Set expected inputs from Indata and aborts with error
                message if input is incorrect
            if (!DA.GetDataList(0, pointList)) return;
            #endregion

            #region Format output
            string BDCString = x + "," + y + "," + z + "," + rx + "," +
                ry + "," + rz;

            for (int i = 0; i < pointList.Count; i++)    //Format
                stringline for all points (identical boundary conditions
                for all points)
            {
                pointInStringFormat.Add(pointList[i].X + "," +
                    pointList[i].Y + "," + pointList[i].Z + ":" +
                    BDCString);
            }
            #endregion

            DA.SetDataList(0, pointInStringFormat);
        } //End of main program

        private List<Point3d> CreatePointList(List<Line> geometry)
        {
            List<Point3d> points = new List<Point3d>();
```

```
            for (int i = 0; i < geometry.Count; i++) //adds every point
                unless it already exists in list
            {
                Line l1 = geometry[i];
                if (!points.Contains(l1.From))
                {
                    points.Add(l1.From);
                }
                if (!points.Contains(l1.To))
                {
                    points.Add(l1.To);
                }
            }
            return points;
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.BDCs;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("c9c208e0-b10b-4ecb-a5ef-57d86a4df109"); }
        }


        /// Component Visual//
        private class Attributes_Custom :
            Grasshopper.Kernel.Attributes.GH_ComponentAttributes
        {
            public Attributes_Custom(GH_Component owner) : base(owner) { }
            protected override void Layout()
            {
                base.Layout();

                Rectangle rec0 = GH_Convert.ToRectangle(Bounds);

                rec0.Height += 42;
```

```
            Rectangle rec1 = rec0;
            rec1.X = rec0.Left + 1;
            rec1.Y = rec0.Bottom - 42;
            rec1.Width = (rec0.Width) / 3 + 1;
            rec1.Height = 22;
            rec1.Inflate(-2, -2);

            Rectangle rec2 = rec1;
            rec2.X = rec1.Right + 2;

            Rectangle rec3 = rec2;
            rec3.X = rec2.Right + 2;

            Rectangle rec4 = rec1;
            rec4.Y = rec1.Bottom + 2;

            Rectangle rec5 = rec4;
            rec5.X = rec4.Right + 2;

            Rectangle rec6 = rec5;
            rec6.X = rec2.Right + 2;

            Bounds = rec0;
            BoundsAllButtons = rec0;
            ButtonBounds = rec1;
            ButtonBounds2 = rec2;
            ButtonBounds3 = rec3;
            ButtonBounds4 = rec4;
            ButtonBounds5 = rec5;
            ButtonBounds6 = rec6;

        }

        GH_Palette xColor = GH_Palette.Black;
        GH_Palette yColor = GH_Palette.Black;
        GH_Palette zColor = GH_Palette.Black;
        GH_Palette rxColor = GH_Palette.Black;
        GH_Palette ryColor = GH_Palette.Black;
        GH_Palette rzColor = GH_Palette.Black;

        private Rectangle BoundsAllButtons { get; set; }
        private Rectangle ButtonBounds { get; set; }
        private Rectangle ButtonBounds2 { get; set; }
```

```
        private Rectangle ButtonBounds3 { get; set; }
        private Rectangle ButtonBounds4 { get; set; }
        private Rectangle ButtonBounds5 { get; set; }
        private Rectangle ButtonBounds6 { get; set; }

        protected override void Render(GH_Canvas canvas, Graphics
            graphics, GH_CanvasChannel channel)
        {
            base.Render(canvas, graphics, channel);
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button =
                    GH_Capsule.CreateTextCapsule(ButtonBounds,
                    ButtonBounds, xColor, "X", 2, 0);
                button.Render(graphics, Selected, Owner.Locked,
                    false);
                button.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button2 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds2,
                    ButtonBounds2, yColor, "Y", 2, 0);
                button2.Render(graphics, Selected, Owner.Locked,
                    false);
                button2.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button3 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds3,
                    ButtonBounds3, zColor, "Z", 2, 0);
                button3.Render(graphics, Selected, Owner.Locked,
                    false);
                button3.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button4 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds4,
                    ButtonBounds4, rxColor, "RX", 2, 0);
                button4.Render(graphics, Selected, Owner.Locked,
                    false);
                button4.Dispose();
```

```
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button5 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds5,
                    ButtonBounds5, ryColor, "RY", 2, 0);
                button5.Render(graphics, Selected, Owner.Locked,
                    false);
                button5.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button6 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds6,
                    ButtonBounds6, rzColor, "RZ", 2, 0);
                button6.Render(graphics, Selected, Owner.Locked,
                    false);
                button6.Dispose();
            }
        }

        public override GH_ObjectResponse
            RespondToMouseDown(GH_Canvas sender, GH_CanvasMouseEvent
            e)
        {
            if (e.Button == MouseButtons.Left)
            {
                RectangleF rec = ButtonBounds;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("X");
                }
                rec = ButtonBounds2;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("Y");
                }
                rec = ButtonBounds3;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("Z");
                }
                rec = ButtonBounds4;
                if (rec.Contains(e.CanvasLocation))
```

```
                {
                    switchColor("RX");
                }
                rec = ButtonBounds5;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("RY");
                }
                rec = ButtonBounds6;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("RZ");
                }
                rec = BoundsAllButtons;
                if (rec.Contains(e.CanvasLocation))
                {
                    if (xColor == GH_Palette.Black) {
                        BDCComponent.setBDC("X", 0); }
                    else  { BDCComponent.setBDC("X", 1); }
                    if (yColor == GH_Palette.Black) {
                        BDCComponent.setBDC("Y", 0); }
                    else  { BDCComponent.setBDC("Y", 1); }
                    if (zColor == GH_Palette.Black) {
                        BDCComponent.setBDC("Z", 0); }
                    else  { BDCComponent.setBDC("Z", 1); }
                    if (rxColor == GH_Palette.Black) {
                        BDCComponent.setBDC("RX", 0); }
                    else { BDCComponent.setBDC("RX", 1); }
                    if (ryColor == GH_Palette.Black) {
                        BDCComponent.setBDC("RY", 0); }
                    else { BDCComponent.setBDC("RY", 1); }
                    if (rzColor == GH_Palette.Black) {
                        BDCComponent.setBDC("RZ", 0); }
                    else { BDCComponent.setBDC("RZ", 1); }
                    Owner.ExpireSolution(true);
                }
                return GH_ObjectResponse.Handled;
            }
            return base.RespondToMouseDown(sender, e);
        }

        private void switchColor(string button)
        {
            if (button == "X")
```

```
                    {
                        if (xColor == GH_Palette.Black) { xColor = GH_Palette.Grey; }
                        else { xColor = GH_Palette.Black; }
                    }
                    else if (button == "Y")
                    {
                        if (yColor == GH_Palette.Black) { yColor = GH_Palette.Grey; }
                        else { yColor = GH_Palette.Black; }
                    }
                    else if (button == "Z")
                    {
                        if (zColor == GH_Palette.Black) { zColor = GH_Palette.Grey; }
                        else { zColor = GH_Palette.Black; }
                    }
                    else if (button == "RX")
                    {
                        if (rxColor == GH_Palette.Black) { rxColor = GH_Palette.Grey; }
                        else { rxColor = GH_Palette.Black; }
                    }
                    else if (button == "RY")
                    {
                        if (ryColor == GH_Palette.Black) { ryColor = GH_Palette.Grey; }
                        else { ryColor = GH_Palette.Black; }
                    }
                    else if (button == "RZ")
                    {
                        if (rzColor == GH_Palette.Black) { rzColor = GH_Palette.Grey; }
                        else { rzColor = GH_Palette.Black; }
                    }
                }
            }
        }
}
```

3D Beam Deformed Geometry Component

```
using System;
```

```
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;
using System.Drawing;
using Grasshopper.GUI.Canvas;
using System.Windows.Forms;
using Grasshopper.GUI;

using MathNet.Numerics.LinearAlgebra;

namespace Beam3D
{
    public class DeformedGeometry : GH_Component
    {
        public DeformedGeometry()
          : base("DeformedGeometry", "DefG",
              "Description",
              "Koala", "3D Beam")
        {
        }

        ////Initialize startcondition
        //static bool startDef = true;


        ////Method to allow C# hanging of variables via GUI (see
            Component Visual)
        //public static void setToggles(string s, bool i)
        //{
        //    if (s == "Color")
        //    {
        //        startDef = i;
        //    }
        //}

        //public override void CreateAttributes()
        //{
        //    m_attributes = new Attributes_Custom(this);
        //}

        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
```

```
        {
            pManager.AddNumberParameter("Stress", "Ss", "Nodal stress",
                GH_ParamAccess.list);
            pManager.AddNumberParameter("Strain", "Sn", "Nodal strain",
                GH_ParamAccess.list);
            pManager.AddGenericParameter("Deformation", "Def",
                "Deformations from 3DBeamCalc", GH_ParamAccess.item);
            pManager.AddPointParameter("New base points", "NBP", "New
                base points from Calc component", GH_ParamAccess.list);
            pManager.AddNumberParameter("Scale", "S", "The Scale Factor
                for Deformation", GH_ParamAccess.item, 1000);
        }

        protected override void
            RegisterOutputParams(GH_Component.GH_OutputParamManager
            pManager)
        {
            pManager.AddNumberParameter("Pure Axial stress", "PA SS",
                "Pure axial stress per sub-element", GH_ParamAccess.list);
            pManager.AddNumberParameter("Pure Axial strain", "PA SN",
                "Pure axial strain per sub-element", GH_ParamAccess.list);
            pManager.AddNumberParameter("Axial stress", "A SS", "Axial
                stress per sub-element", GH_ParamAccess.list);
            pManager.AddNumberParameter("Axial strain", "A SN", "Axial
                strain per sub-element", GH_ParamAccess.list);
            pManager.AddCurveParameter("Deformed Geometry", "Def.G.",
                "Deformed Geometry as List of Lines",
                GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch
            //Expected inputs and outputs
            List<Curve> defC = new List<Curve>();
            List<double> stress = new List<double>();
            List<double> strain = new List<double>();
            Matrix<double> def = Matrix<double>.Build.Dense(1, 1);
            List<Point3d> oldXYZ = new List<Point3d>();
            double scale = 1000; //input deformation scale


            //Set expected inputs from Indata
            if (!DA.GetDataList(0, stress)) return;
```

```
            if (!DA.GetDataList(1, strain)) return;
            if (!DA.GetData(2, ref def)) return;
            if (!DA.GetDataList(3, oldXYZ)) return;
            if (!DA.GetData(4, ref scale)) return;
            #endregion

            #region Deformed geometry
            //no. of sub-nodes per main element
            int n = def.ColumnCount / 6;
            //number of sub-elements
            int ns = n - 1;

            //scale deformations
            def = scale * def;

            if (oldXYZ.Count == 0) return;
            //Calculate new nodal points
            for (int i = 0; i < def.RowCount; i++)
            {
                List<Point3d> tempNew = new List<Point3d>();
                for (int j = 0; j < n; j++)
                {
                    //original xyz
                    var tP = oldXYZ[i * n + j];

                    //add deformations
                    tP.X = tP.X + def[i, j * 6];
                    tP.Y = tP.Y + def[i, j * 6 + 1];
                    tP.Z = tP.Z + def[i, j * 6 + 2];

                    //replace previous xyz with displaced xyz
                    tempNew.Add(tP);
                }
                //Create Curve based on new nodal points(degree = 3)
                Curve nc = Curve.CreateInterpolatedCurve(tempNew, 3);
                defC.Add(nc);
            }
            #endregion

            List<double> ss_x = new List<double>();
            List<double> sn_x = new List<double>();
            List<double> ss_y = new List<double>();
            List<double> sn_y = new List<double>();
            List<double> ss_z = new List<double>();
```

```
            List<double> sn_z = new List<double>();

            for (int i = 0; i < stress.Count / 3; i++)
            {
                ss_x.Add(stress[i * 3]);
                sn_x.Add(strain[i * 3]);
                ss_y.Add(stress[i * 3 + 1]);
                sn_y.Add(strain[i * 3 + 1]);
                ss_z.Add(stress[i * 3 + 2]);
                sn_z.Add(strain[i * 3 + 2]);
            }

            ss_x = GetAverage(ss_x, ns, defC.Count);
            sn_x = GetAverage(sn_x, ns, defC.Count);
            ss_y = GetAverage(ss_y, ns, defC.Count);
            sn_y = GetAverage(sn_y, ns, defC.Count);
            ss_z = GetAverage(ss_z, ns, defC.Count);
            sn_z = GetAverage(sn_z, ns, defC.Count);

            List<double> ss = new List<double>();
            List<double> sn = new List<double>();

            for (int i = 0; i < ss_x.Count; i++)
            {
                if (ss_x[i] > 0)
                {
                    ss.Add(ss_x[i] + Math.Abs(ss_y[i]) +
                        Math.Abs(ss_z[i]));
                    sn.Add(ss_x[i] + Math.Abs(sn_y[i]) +
                        Math.Abs(sn_z[i]));
                }
                else
                {
                    ss.Add(ss_x[i] - Math.Abs(ss_y[i]) -
                        Math.Abs(ss_z[i]));
                    sn.Add(sn_x[i] - Math.Abs(sn_y[i]) -
                        Math.Abs(sn_z[i]));
                }
            }



            DA.SetDataList(0, ss_x);
            DA.SetDataList(1, sn_x);
```

```
            DA.SetDataList(2, ss);
            DA.SetDataList(3, sn);
            DA.SetDataList(4, defC);
        }//End of main program

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.Draw;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("6391b902-2ec8-487c-94fd-b921479620b3"); }
        }

        private List<double> GetAverage(List<double> s, int n, int el)
        {
            var s_avg = new List<double>();
            for (int i = 0, ct = 0; s_avg.Count < el*n; i++)
            {
                if (ct == n)
                {
                    ct = 0;
                    continue;
                }
                s_avg.Add((s[i] + s[i + 1]) / 2);
                ct++;
            }
            return s_avg;
        }
    }
}
```

Appendix D

Shell

D.1 Local axes and direction cosine

Matlab code for generating local axes from a triangular element, calculation the directional cosines and a function for exporting the direction cosine as C# code. With transformation of a triangle from global to local coordinates, graphs and example code.

```
clear;
syms ax ay az bx by bz;
syms x1 y1 z1 x2 y2 z2 x3 y3 z3;

% Cross product gives vector perpendicualr to both vectors

cx = ay*bz - az*by;
cy = az*bx - ax*bz;
cz = ax*by - ay*bx;

ax = x2-x1;
ay = y2-y1;
az = z2-z1;

bx = x3-x1;
by = y3-y1;
bz = z3-z1;
```

```
cx = subs(cx);
cy = subs(cy);
cz = subs(cz);

% a is now x-axis and c is z axis, need to find y-axis as b

clear('bx','by','bz');

syms bx by bz;

bx = cy*az - cz*ay;
by = cz*ax - cx*az;
bz = cx*ay - cy*ax;

a = [ax ay az];
b = [bx by bz];
c = [cx cy cz];

Lx = sqrt(ax^2 + ay^2 + az^2);
Ly = sqrt(bx^2 + by^2 + bz^2);
Lz = sqrt(cx^2 + cy^2 + cz^2);

x = [x1 x2 y1 y2 z1 z2];
y = [x1 bx+x1 y1 by+y1 z1 bz+z1];
z = [x1 cx+x1 y1 cy+y1 z1 cz+z1];

cosxX = (ax)/Lx;
cosxY = (ay)/Lx;
cosxZ = (az)/Lx;
cosyX = (bx)/Ly;
cosyY = (by)/Ly;
cosyZ = (bz)/Ly;
coszX = (cx)/Lz;
coszY = (cy)/Lz;
coszZ = (cz)/Lz;

s = [cosxX cosxY cosxZ cosyX cosyY cosyZ coszX coszY coszZ];
```

```
exportCosXX( s, 'cosxX.txt');

% Testing by inserting values

runtest = 1;

if runtest
    x1 = 0;
    x2 = 2125;
    x3 = 0;

    y1 = 0;
    y2 = 0;
    y3 = 2382.5;

    z1 = 0;
    z2 = 1827;
    z3 = 1358;

    m = [x1 x2 x3;y1 y2 y3;z1 z2 z3];

    ax = double(subs(ax));
    ay = double(subs(ay));
    az = double(subs(az));
    bx = double(subs(bx));
    by = double(subs(by));
    bz = double(subs(bz));
    cx = double(subs(cx));
    cy = double(subs(cy));
    cz = double(subs(cz));

    Lx = double(subs(Lx));
    Ly = double(subs(Ly));
    Lz = double(subs(Lz));

    a = [ax ay az]./Lx;
    b = [bx by bz]./Ly;
```

```
    c = [cx cy cz]./Lz;

    x = [x1 (a(1)+x1) y1 (a(2)+y1) z1 (a(3)+z1)];
    y = [x1 (bx/Ly+x1) y1 (by/Ly+y1) z1 (bz/Ly+z1)];
    z = [x1 (cx/Lz+x1) y1 (cy/Lz+y1) z1 (cz/Lz+z1)];

    Lx = sqrt(a(1)^2 + a(2)^2 + a(3)^2);
    Ly = sqrt((y(2)-y(1))^2 + (y(4)-y(3))^2 + (y(6)-y(5))^2);
    Lz = sqrt((z(2)-z(1))^2 + (z(4)-z(3))^2 + (z(6)-z(5))^2);

    % dot product of two vectors should be 0 when perpendicular

    dotxy = a(1)*b(1) + a(2)*b(2) + a(3)*b(3);
    if round(dotxy,10) == 0
        fprintf('x and y axis are perpendicular, OK!\n');
    else
        fprintf('x and y axis are NOT perpendicular..\n');
    end
    dotxz = a(1)*c(1) + a(2)*c(2) + a(3)*c(3);
    if round(dotxz,10) == 0
        fprintf('x and z axis are perpendicular, OK!\n');
    else
        fprintf('x and z axis are NOT perpendicular..\n');
    end
    dotyz = b(1)*c(1) + b(2)*c(2) + b(3)*c(3);
    if round(dotyz,10) == 0
        fprintf('y and z axis are perpendicular, OK!\n');
    else
        fprintf('y and z axis are NOT perpendicular..\n');
    end

    if round(Lx,10) == 1
        fprintf('Length of x is OK!\n')
    else
        fprintf('Length of x is NOT ok!\n')
    end
    if round(Ly,10) == 1
        fprintf('Length of y is OK!\n')
```

```
    else
        fprintf('Length of y is NOT ok!\n')
    end
    if round(Lz,10) == 1
        fprintf('Length of z is OK!\n')
    else
        fprintf('Length of z is NOT ok!\n')
    end

    figure;
    plot3(x(1:2),x(3:4),x(5:6));
    hold on
    plot3(y(1:2),y(3:4),y(5:6));
    plot3(z(1:2),z(3:4),z(5:6));
    plot3([x1 x2 x3 x1],[y1 y2 y3 y1],[z1 z2 z3 z1]);
    grid on
    rotate3d on
    pbaspect([1 1 1]);
    title('3D graph of triangle in global coordinates');

    T = zeros(3,3);
    T(1,1:3) = subs(s(1:3));
    T(2,1:3) = subs(s(4:6));
    T(3,1:3) = subs(s(7:9));
    T = double(T);

    ml = T*m;
    al = T*transpose(a);
    bl = T*transpose(b);
    cl = T*transpose(c);

    xl = [ml(1,1) (al(1)/Lx+ml(1,1)) ml(2,1) (al(2)/Lx+ml(2,1))
        ml(3,1) (al(3)/Lx+ml(3,1))];
    yl = [ml(1,1) (bl(1)/Ly+ml(1,1)) ml(2,1) (bl(2)/Ly+ml(2,1))
        ml(3,1) (bl(3)/Ly+ml(3,1))];
    zl = [ml(1,1) (cl(1)/Lz+ml(1,1)) ml(2,1) (cl(2)/Lz+ml(2,1))
        ml(3,1) (cl(3)/Lz+ml(3,1))];

```

```
    xl = xl - [xl(1) xl(1) xl(3) xl(3) xl(5) xl(5)];
    yl = yl - [yl(1) yl(1) yl(3) yl(3) yl(5) yl(5)];
    zl = zl - [zl(1) zl(1) zl(3) zl(3) zl(5) zl(5)];
    ml = ml- repmat(ml(:,1),[1,3]);

    figure;
    plot3(xl(1:2),xl(3:4),xl(5:6));
    hold on
    plot3(yl(1:2),yl(3:4),yl(5:6));
    plot3(zl(1:2),zl(3:4),zl(5:6));
    plot3([ml(1,1) ml(1,2) ml(1,3) ml(1,1)],[ml(2,1)...
        ml(2,2) ml(2,3) ml(2,1)],[ml(3,1) ml(3,2)...
        ml(3,3) ml(3,1)]);
    grid on
    rotate3d on
    pbaspect([1 1 1]);
    title('3D graph of triangle in local coordinates with
        z-axis');

    figure;
    plot(ml(1,:),ml(2,:),[ml(1,3) ml(1,1)],[ml(2,3) ml(2,1)]);
    title('Triangle in x and y local coordinates');
    hold on
    plot([ml(1,1) ml(1,1)+al(1)],[ml(2,1) ml(2,1)+al(2)]);
    plot([ml(1,1) ml(1,1)+bl(1)],[ml(2,1) ml(2,1)+bl(2)]);

%     figure;
%     plot([ml(1,1) ml(1,1)+al(1)],[ml(3,1) ml(3,1)+al(3)]);
%     hold on}end
```

```

function [ ] = exportCosXX( s, txt )

temptxt = {};
temptxtindex = 1;
for i = 1:1
    for j= 1:9
        jt = num2str(j - 1);
```

```
s1 = strcat('cosxX =');
s2 = char(s(j));
hatt = strfind(s2,'^');
s2temp = s2;
placements = [];
power = [];

for l = hatt
    power = [s2(l+1) power];
    count = 0;
    hasChanged = false;
    found = false;
    rev = 1;
    pos = l - rev;
    while found == false
        if s2(pos) == ')'
            count = count + 1;
        elseif s2(pos) == '('
            count = count - 1;
        elseif count == 0 && (isletter(s2(pos)) ||
            isempty(str2num(s2(pos))) == 0)
            variablecount = 0;
            if (isletter(s2(pos-1)) ||
                isempty(str2num(s2(pos-1))) == 0)
                variablecount = 1;
                if (isletter(s2(pos-2)) ||
                    isempty(str2num(s2(pos-2))) == 0)
                    variablecount = 2;
                    if (isletter(s2(pos-3)) ||
                        isempty(str2num(s2(pos-3))) ==
                        0)
                        variablecount = 3;
                    end
                end
            end
            pos = pos - variablecount;
            break;
        end
```

```
                if hasChanged == false && count > 0
                    hasChanged = true;
                elseif hasChanged == true && count == 0
                    found = true;
                end
                if found == false
                    pos = pos - rev;
                end
            end
            placements = [placements pos];
        end

        plsr = fliplr(placements);
        hattr = fliplr(hatt);
        plsr = sort(plsr,'descend');
        while (isempty(plsr)==0 || isempty(hattr)==0)
            if isempty(hattr) || plsr(1) > hattr(1)
                s2temp = strcat(s2temp(1:plsr(1)-1),...
                'Math.Pow(',s2temp(plsr(1):end));
                plsr = plsr(2:end);
            else
                s2temp = strcat(s2temp(1:hattr(1)-1),...
                ',',s2temp(hattr(1)+1),...
                    ')',s2temp(hattr(1)+2:end));
                hattr = hattr(2:end);
            end
        end
        if ~strcmp(s2temp,'0')
            temptxt(temptxtindex) = strcat(s1,{' '},s2temp,';');
            temptxtindex = temptxtindex + 1;
        end

    end

    fid = fopen(txt,'w');
    for i = 1:length(temptxt)
        fprintf(fid,'%s\n',char(temptxt(i)));
    end
```

```
    fclose(fid);
end
```

D.2 Derivation of element stiffness matrix for CST and Morley

```

clear;
%
%
% ------------------------------------------ START BENDING TRIANGLE
%
%
%NB numbering counter clockwise!
%
% ^ y-axis          o 3
% |                / \                 The Morley triangle, the
% |               /   \                simplest bending triangle
% |              /     \               possible!
% |         6 o         o 5            node 4,5,6 have only
% |          /           \             rotation along the
% |         /     ^n4     \            triangle edge. phi6
% |        /      |        \           rotates the axis from 1 to
% |   1 o - - - >> -> - o 2            3, while phi4 from 1 to 2
% |               4    t4              and 5 from 3 to 2. As shown
% |                                    for node 4. Each of these
% |       vertices has their owncoordinate system n(perpendivcular
% |       to the vertice, upwards. The angle between the local
% |       t-axis and the x-axis is denoted a (ie. a(1) for node 4)
% |
% |
% o - - - - - - - - -> x-axis
%
% v = [w1 w2 w3 phi4 phi5 phi6]
%
% x = c*t - s*n       c = cos(a), s = sin(a)
% y = s*t + c*n
% t = c*x + s*y
% n = -s*x + c*y
%
```

```
% df/dt = c*df/dx + s*df/dy
% df/dn = -s*df/dx + c*df/dy
%
% df/dx = 1/2A * (df/dxi1 * y23 + df/dxi2 * y31 + df/dxi3 * y12)
% df/dy = 1/2A * (df/dxi1 * x32 + df/dxi2 * x13 + df/dxi3 * x21)
% (ref FEA Bell s149)
%
% Bq = Delta*Nq
%
%       [ -z*d^2/dx^2*N1 , -z*d^2/dx^2*N2 , ... -z*d^2/dx^2*Nn ]
% Bq =  [ -z*d^2/dy^2*N1 , -z*d^2/dy^2*N2 , ... -z*d^2/dy^2*Nn ]
%       [ -2z*d^2/dxdy*N1 , -2z*d^2/dxdy*N2 , ...
    -2z*d^2/dxdy*Nn]
%
% B = Bq * A^-1
% (ref FEA Bell s167)
%

syms xi1 xi2 xi3 xi4 xi5 xi6;

%xi3 = 1 - xi1 - xi2;

Nq = [xi1^2 xi2^2 xi3^2 xi1*xi2 xi2*xi3 xi3*xi1];

syms w1 w2 w3 phi1 phi2 phi3;
syms xi1 xi2 xi3;
syms x1 y1 x2 y2 x3 y3 z;
syms Area t;
syms x13 x21 x32 y12 y23 y31;
syms E nu;
syms c4 c5 c6 s4 s5 s6 ga4 ga5 ga6 my4 my5 my6 a4 a5 a6;

s = [s4 s5 s6];
c = [c4 c5 c6];
ga = [ga4 ga5 ga6];
my = [my4 my5 my6];
a = [a4 a5 a6];
% a(1) = ga(1) + my(1);
```

```
% a(2) = ga(2) + my(2);
% a(3) = ga(3) + my(3);

A21 = [ga(1) my(1) 0;0 my(2) -a(2);ga(3) 0 -a(3)];
A22 = 1/2 * [a(1) -a(1) -a(1);ga(2) -ga(2) ga(2);my(3) my(3)
    -my(3)];
I = [1 0 0;0 1 0;0 0 1];
A = [I,zeros(3,3);A21 A22];
A_inv = [I zeros(3,3);-inv(A22)*A21 inv(A22)];

% Pricinpal usage of the A matrix:
%             | 1  0  0 |
% --> A^-1 = | 0  1  0 |
%             | 0  0  1 |
%               |  |  |
%               |  |  |
%               V  V  V
%              N1 N2 N3
% N1 = xi1
% N2 = xi2
% N3 = xi3

% Create shape functions
for i = 1:length(A_inv)
    Nb(i) = A_inv(1,i)*Nq(1) + A_inv(2,i)*Nq(2) +
        A_inv(3,i)*Nq(3) + ...
         A_inv(4,i)*Nq(4) + A_inv(5,i)*Nq(5) + A_inv(6,i)*Nq(6);
    Nbt(i) = A_inv(1,i)*Nq(1) + A_inv(2,i)*Nq(2) +
        A_inv(3,i)*Nq(3);
end

% syms B11 B12 B13 B14 B15 B16;
% syms B21 B22 B23 B24 B25 B26;
% syms B31 B32 B33 B34 B35 B36;
%
%
% Bq = [B11 B12 B13 B14 B15 B16;B21 B22 B23 B24 B25 B26;B31 B32
    B33 B34 B35 B36];
```

```
%
%

% Bq = [diff(Nq,xi1,2);diff(Nq,xi2,2);diff(diff(Nq,xi1),xi2)];
% Bq = double(Bq);

Bq = [2 0 2 0 0 -2;0 2 2 0 -2 0;0 0 2 1 -1 -1];


 H_additional = 1/(4*Area^2);
H = [y23^2 y31^2 2*y31*y23; x32^2 x13^2 2*x13*x32; ...
     2*x32*y23 2*x13*y31 2*(x13*y23+x32*y31)];

Bb = H*Bq*A_inv;
% Bb = Bb * H_additional;
Bb_T = transpose(Bb);
exportKmatrix(simplify(Bb),'Bk_b','Bk_b.txt')

 C_additional = E/(1-nu^2);
C = [1 nu 0;nu 1 0;0 0 (1-nu)/2];
exportKmatrix(simplify(C),'C','C.txt')
% (ref FEA Bell s85)
%
% Further k = B^T*C*B*Area*t
% (ref FEA Bell s167/127)

 k_additional = (Area*t^3)/12;
kb = Bb_T*C*Bb;
kb = k_additional*C_additional*H_additional^2 *kb;

%
%
% ------------------------------ END BENDING TRIANGLE
%
%
%
%
%
```

```
% ---------------- START CONSTANT STRAIN/STRESS TRIANGLE
%
%
% Now we will look at a plane triangle with deformation
% in x and y direction in node 1, 2 and 3.
% For simplicity we will use the simplest triangle,
% the Constant Strain Triangle (CST).
% This gives us thus 6 dofs.
%
%                                                    [ u1 ]
%                                                    [ v1 ]
%   U = [ u ] = [ xi1   0   xi2   0   xi3   0 ] = [ u2 ]
%       [ v ]   [  0   xi1   0   xi2   0   xi3]   [ v2 ]
%                                                    [ u3 ]
%                                                    [ v3 ]
%
% u = a1 + a2*x + a3*y
% v = b1 + b2*x + b3*y
%
% strains: ex = a2, ey = b3, yxy = a3 + b2 (constant)
%
% u1 = a1 + a2*x1 + a3*y1
% u2 = a2 + a2*x2 + a3*y2
% u2 = a3 + a2*x3 + a3*y3
%
% v1 = b1 + b2*x1 + b3*y1
% v2 = b2 + b2*x2 + b3*y2
% v3 = b3 + b2*x3 + b3*y3
%
%                              (u1)
%                              (v1)
% (u)     [ N1   0 N2   0 N3   0 ]{u2)
% (v) = [ 0   N1 0   N2 0   N3 ]{v2)
%                              (u3)
%                              (v3)
%
syms x y x1 x2 x3 y1 y2 y3;

```

```
u1 = [1 x1 y1];
u2 = [1 x2 y2];
u3 = [1 x3 y3];

v1 = [1 x1 y1];
v2 = [1 x2 y2];
v3 = [1 x3 y3];

Am = [ u1 0 0 0 ; 0 0 0 v1 ; u2 0 0 0 ; 0 0 0 v2 ; u3 0 0 0 ;
    0 0 0 v3];
Am_inv_additional = (x1*y2 - x2*y1 - x1*y3 + x3*y1 + x2*y3 -
    x3*y2);
Am_inv = inv(Am)*Am_inv_additional;

Nm = sym([]);
for i = 1:3
%     for j = 1:length(A)
        Nm(i) = (Am_inv(1,i*2-1) + Am_inv(2,i*2-1)*x + ...
            Am_inv(3,i*2-1)*y + Am_inv(4,i*2-1) +
                Am_inv(5,i*2-1)*x +...
            Am_inv(6,i*2-1)*y)/Am_inv_additional;
%     end
end

Bm = sym([]);
for i = 1:3
    temp = [diff(Nm(i),x) 0 ; 0 diff(Nm(i),y) ; ...
        diff(Nm(i),y) diff(Nm(i),x)];
    Bm = [Bm temp];
end

exportKmatrix(Bm,'Bk_m','Bk_m.txt');

Bm_t = transpose(Bm);
km = Bm_t*C*Bm*C_additional*Area*t;

%
%
```

```
% ------------------------------------------ END CONSTANT STRAIN/STRESS TRIANGLE
%
%

[m,n] = size(km);
[g,h] = size(kb);

k = [km zeros(m,h); zeros(g,n) kb];

% sort k matrix by [x1 y1 w1 phi1 x2 y2 w2 phi2 x3 y3 w3 phi3]
%k = k([1 2 7 10 3 4 8 11 5 6 9 12], [1 2 7 10 3 4 8 11 5 6 9 12]);
%exportKmatrix(simplify(k),'kmatrix.txt')

% Test 1(0,0,0) 2(4,1,0) 3(2,5,0)
if 1
    x1 = -2020;
    x2 = 0;
    x3 = 0;

    y1 = -4000;
    y2 = -4000;
    y3 = 0;

    z1 = 0;
    z2 = 0;
    z3 = 0;

    x4 = x1+(x2-x1)/2;
    x5 = x2+(x3-x2)/2;
    x6 = x3+(x3-x1)/2;

    y4 = y1+(y2-y1)/2;
    y5 = y2+(y3-y2)/2;
    y6 = y3+(y3-y1)/2;
    x13 = x1 - x3;
    x21 = x2 - x1;
    x32 = x3 - x2;
```

```
    y12 = y1 - y2;
    y23 = y2 - y3;
    y31 = y3 - y1;
    xu = [x1 x2 x3 x1];
    yu = [y1 y2 y3 y1];
    t = 10;
    Area = abs((x1*(y2-y3)+x2*(y3-y1)+x3*(y1-y2))/2);
    nu = 0.3;
    E = 200000;

for m = [1,2,3]
    L(m) = sqrt((xu(m+1)-xu(m))^2+(yu(m+1)-yu(m))^2);
    if xu(m+1)>xu(m)
        c(m)=(xu(m+1)-xu(m))/L(m);
        s(m)=(yu(m+1)-yu(m))/L(m);
    elseif xu(m+1)<xu(m)
        c(m)=(xu(m)-xu(m+1))/L(m);
        s(m)=(yu(m)-yu(m+1))/L(m);
    else
        c(m)=0;
        s(m)=1;
    end

    ga(m) = (c(m)*x32-s(m)*y23)/(2*Area);
    my(m) = (c(m)*x13-s(m)*y31)/(2*Area);
    a(m) = ga(m) + my(m);
end

ga4 = double(ga(1));
ga5 = double(ga(2));
ga6 = double(ga(3));
my4 = double(my(1));
my5 = double(my(2));
my6 = double(my(3));
a4 = ga4 + my4;
a5 = ga5 + my5;
a6 = ga6 + my6;
```

```
Bb = double(subs(Bb));
Bm = double(subs(Bm));
k = double(subs(k));
kb = double(subs(kb));
km = double(subs(km));

% displ = [0 0 1 0 1 1];
% R = [0 0 -0.5 0 0.5 1];
%
% for i = 1:length(km)
%     if displ(i) == 0
%         km(i,:) = 0;
%         km(:,i) = 0;
%         km(i,i) = 1;
%     end
% end


% plot([x1 x2 x3 x1],[y1 y2 y3 y1])
% hold on
% old =[x1 y1 x2 y2 x3 y3];
% grid on
% new = [x1 y1 x2 y2 x3 y3]+res';
% plot([new(1) new(3) new(5) new(1)],[new(2) new(4) new(6)
    new(2)])
%
% x = 0.4;
% y = 0.2;
% xi1 = 0.4;
% xi2 = 0.4;
% xi3 = 1 - xi1 - xi2;
%
% Nm = subs(Nm);
% Nb = subs(Nb(1:3));
% fprintf('Shapefunction membrane sum: %f\n',sum(Nm))
% fprintf('Shapefunction bending sum: %f\n',sum(Nb))
end
```

```
function [ ] = exportKmatrix( k, s, txt )

[m,n] = size(k);
temptxt = {};
temptxtindex = 1;
for i = 1:m
    for j= 1:n
        it = num2str(i - 1);
        jt = num2str(j - 1);
        s1 = strcat(s,'[',it,',',jt,'] =');
        s2 = char(k(i,j));
        hatt = strfind(s2,'^');
        s2temp = s2;
        placements = [];
        power = [];

        for l = hatt
            power = [s2(l+1) power];
            count = 0;
            hasChanged = false;
            found = false;
            rev = 1;
            pos = l - rev;
            while found == false
                if s2(pos) == ')'
                    count = count + 1;
                elseif s2(pos) == '('
                    count = count - 1;
                elseif count == 0 && (isletter(s2(pos)) ||
                    isempty(str2num(s2(pos))) == 0)
                    variablecount = 0;
                    if (isletter(s2(pos-1)) ||
                        isempty(str2num(s2(pos-1))) == 0)
                        variablecount = 1;
                        if (isletter(s2(pos-2)) ||
                            isempty(str2num(s2(pos-2))) == 0)
                            variablecount = 2;
                            if (isletter(s2(pos-3)) ||
```

```
                                        isempty(str2num(s2(pos-3))) ==
                                        0)
                                        variablecount = 3;
                                    end
                                end
                            end
                            pos = pos - variablecount;
                            break;
                        end
                        if hasChanged == false && count > 0
                            hasChanged = true;
                        elseif hasChanged == true && count == 0
                            found = true;
                        end
                        if found == false
                            pos = pos - rev;
                        end
                    end
                    placements = [placements pos];
                end
%                   tel = 1;
                plsr = fliplr(placements);
                hattr = fliplr(hatt);
%                   for it = plsr
%                       if hattr(1) >= length(s2temp)+2 && it == plsr(1)
%                           s2temp = strcat(s2temp(1:it),'Math.Pow(',...
%                           s2temp(it:hattr(tel)-1),',',power(tel),')');
%                       else
%                           s2temp = strcat(s2temp(1:it-1),'Math.Pow(',...
%                           s2temp(it:hattr(tel)-1),',',power(tel),')',...
%                           s2temp(hattr(tel)+2:end));
%                       end
%                       tel = tel + 1;
%                   end
                plsr = sort(plsr,'descend');
                while (isempty(plsr)==0 || isempty(hattr)==0)
                    if isempty(hattr) || plsr(1) > hattr(1)
                        s2temp = strcat(s2temp(1:plsr(1)-1),...
```

```
                    'Math.Pow(',s2temp(plsr(1):end));
                    plsr = plsr(2:end);
                else
                    s2temp = strcat(s2temp(1:hattr(1)-1),...
                    ',',s2temp(hattr(1)+1),...
                    ')',s2temp(hattr(1)+2:end));
                    hattr = hattr(2:end);
                end
            end
            if ~strcmp(s2temp,'0')
                temptxt(temptxtindex) = strcat(s1,{' '},s2temp,';');
                temptxtindex = temptxtindex + 1;
            end

    end

    fid = fopen(txt,'w');
    for j = 1:length(temptxt)
        fprintf(fid,'%s\n',char(temptxt(j)));
    end
    fclose(fid);
end
```

D.3 Shell source code

Shell Calculation Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using System.Drawing;
using Grasshopper.GUI.Canvas;
using System.Windows.Forms;
using Grasshopper.GUI;

using MathNet.Numerics.LinearAlgebra;
using MathNet.Numerics.LinearAlgebra.Double;
using System.Diagnostics;
using Rhino.Geometry;

namespace Shell
{
    public class ShellComponent : GH_Component
    {
        public ShellComponent()
          : base("ShellCalculation", "SC",
              "Description",
              "Koala", "Shell")
        {
        }

        static bool startCalc = false;

        public static void setStart(string s, bool i)
        {
            if (s == "Run")
            {
                startCalc = i;
            }
        }

        public override void CreateAttributes()
        {
            m_attributes = new Attributes_Custom(this);
        }

```

```
        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddMeshParameter("Mesh", "M", "The Meshed shell
                structure", GH_ParamAccess.item);
            pManager.AddTextParameter("Boundary Conditions", "BDC",
                "Boundary Conditions in form x,y,z,vx,vy,vz,rx,ry,rz",
                GH_ParamAccess.list);
            pManager.AddTextParameter("Material Properties", "Mat",
                "Material Properties: E,v,t,G", GH_ParamAccess.item,
                "200000,0.3,10");
            pManager.AddTextParameter("Point Loads", "PL", "Load given as
                Vector [N]", GH_ParamAccess.list);
        }

        protected override void
            RegisterOutputParams(GH_Component.GH_OutputParamManager
            pManager)
        {
            pManager.AddNumberParameter("Deformations", "Def",
                "Deformations", GH_ParamAccess.list);
            pManager.AddNumberParameter("Reaction Forces", "R", "Reaction
                Forces", GH_ParamAccess.list);
            pManager.AddNumberParameter("Element Stresses", "Strs", "The
                Stress in each element", GH_ParamAccess.list);
            pManager.AddNumberParameter("Element Strains", "Strn", "The
                Strain in each element", GH_ParamAccess.list);
            //pManager.AddTextParameter("Part Timer", "", "",
                GH_ParamAccess.item);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch inputs and assign to variables

            //Expected inputs
            Mesh mesh = new Mesh();                       //mesh in
                Mesh format
            List<MeshFace> faces = new List<MeshFace>();    //faces of
                mesh as a list
            List<Point3d> vertices = new List<Point3d>();   //vertices of
                mesh as a list

```

```
        List<string> bdctxt = new List<string>();       //Boundary
            conditions in string format
        List<string> loadtxt = new List<string>();      //loads in
            string format
        List<string> momenttxt = new List<string>();    //Moments in
            string format
        string mattxt = "";                             //Material in
            string format

        if (!DA.GetData(0, ref mesh)) return;           //sets
            inputted mesh into variable
        if (!DA.GetDataList(1, bdctxt)) return;         //sets
            boundary conditions as string
        if (!DA.GetData(2, ref mattxt)) return;         //sets
            material properties as string
        if (!DA.GetDataList(3, loadtxt)) return;        //sets load
            as string

        foreach (var face in mesh.Faces)
        {
            faces.Add(face);
        }

        foreach (var vertice in mesh.Vertices)
        {
            Point3d temp_vertice = new Point3d();
            temp_vertice.X = Math.Round(vertice.X, 4);
            temp_vertice.Y = Math.Round(vertice.Y, 4);
            temp_vertice.Z = Math.Round(vertice.Z, 4);
            vertices.Add(temp_vertice);
        }

        // Number of edges from Euler's formula
        int NoOfEdges = vertices.Count + faces.Count - 1;
        List<Line> edges = new List<Line>(NoOfEdges);
        #region Create edge list
        Vector<double> nakedEdge =
            Vector<double>.Build.Dense(NoOfEdges,1);
        foreach (var face in faces)
        {
            Point3d vA = vertices[face.A];
            Point3d vB = vertices[face.B];
            Point3d vC = vertices[face.C];
            Line lineAB = new Line(vA, vB);
```

```
            Line lineBA = new Line(vB, vA);
            Line lineCB = new Line(vC, vB);
            Line lineBC = new Line(vB, vC);
            Line lineAC = new Line(vA, vC);
            Line lineCA = new Line(vC, vA);

            if (!edges.Contains(lineAB) && !edges.Contains(lineBA))
            {
                edges.Add(lineAB);
            }
            else
            {
                int i = edges.IndexOf(lineAB);
                if (i == -1)
                {
                    i = edges.IndexOf(lineBA);
                }
                nakedEdge[i] = 0;
            }
            if (!edges.Contains(lineCB) && !edges.Contains(lineBC))
            {
                edges.Add(lineBC);
            }
            else
            {
                int i = edges.IndexOf(lineBC);
                if (i == -1)
                {
                    i = edges.IndexOf(lineCB);
                }
                nakedEdge[i] = 0;
            }
            if (!edges.Contains(lineAC) && !edges.Contains(lineCA))
            {
                edges.Add(lineAC);
            }
            else
            {
                int i = edges.IndexOf(lineAC);
                if (i == -1)
                {
                    i = edges.IndexOf(lineCA);
                }
                nakedEdge[i] = 0;
```

```
    }
}
#endregion

List<Point3d> uniqueNodes;
GetUniqueNodes(vertices, out uniqueNodes);
int gdofs = uniqueNodes.Count * 3 + edges.Count;

//Interpret and set material parameters
double E;       //Material Young's modulus, initial value 210000 [MPa]
double G;       //Shear modulus, initial value 79300 [mm^4]
double nu;      //Poisson's ratio, initially 0.3
double t;       //Thickness of shell
SetMaterial(mattxt, out E, out G, out nu, out t);

#endregion

Vector<double> def_tot;
Vector<double> reactions;
Vector<double> internalStresses;
Vector<double> internalStrains;
List<double> reac = new List<double>();
Matrix<double> K_red;
Vector<double> load_red;
Vector<double> MorleyMoments = Vector<double>.Build.Dense(faces.Count * 3);

#region Prepares boundary conditions and loads for calculation

//Interpret the BDC inputs (text) and create list of boundary condition (1/0 = free/clamped) for each dof.
Vector<double> bdc_value = CreateBDCList(bdctxt, uniqueNodes, faces, vertices, edges);

Vector<double> nakededge = Vector<double>.Build.Dense(gdofs, 0);
for (int i = uniqueNodes.Count*3; i < gdofs; i++)
{
    if (bdc_value[i] == 1)
    {
        nakededge[i] = (nakedEdge[i - uniqueNodes.Count * 3]);
    }
}
```

```
            List<double> test1 = new List<double>(nakededge.ToArray());

            //Interpreting input load (text) and creating load list
                (double)
            List<double> load = CreateLoadList(loadtxt, momenttxt,
                uniqueNodes, faces, vertices, edges);
            #endregion

            if (startCalc)
            {
                #region Create global and reduced stiffness matrix

                //Create global stiffness matrix

                Matrix<double> B; // all B_k matrices collected
                List<int> BOrder; //
                Matrix<double> K_tot;
                //GlobalStiffnessMatrix(faces, vertices, edges,
                    uniqueNodes, gdofs, E, A, Iy, Iz, J, G, nu, t, out
                    K_tot, out B, out BOrder);
                GlobalStiffnessMatrix(faces, vertices, edges,
                    uniqueNodes, gdofs, E, G, nu, t, out K_tot, out B,
                    out BOrder);

                //Create reduced K-matrix and reduced load list (removed
                    clamped dofs)
                CreateReducedGlobalStiffnessMatrix(bdc_value, K_tot,
                    load, uniqueNodes, nakededge, out K_red, out
                    load_red);

                #endregion


                #region Calculate deformations, reaction forces and
                    internal strains and stresses

                //Calculate deformations

                Vector<double> def_reduced =
                    Vector<double>.Build.Dense(K_red.ColumnCount);
                def_reduced = K_red.Cholesky().Solve(load_red);

                //Add the clamped dofs (= 0) to the deformations list
                def_tot = RestoreTotalDeformationVector(def_reduced,
```

```
                    bdc_value, nakededge);

                //Calculate the reaction forces from the deformations
                reactions = K_tot.Multiply(def_tot);

                // strains and stresses as [eps_x eps_y gamma_xy eps_xb
                    eps_yb gamma_xyb ... repeat for each face...]^T b for
                    bending
                CalculateInternalStrainsAndStresses(def_tot, vertices,
                    faces, B, BOrder, uniqueNodes, edges, E, t, nu, out
                    internalStresses, out internalStrains, out
                    MorleyMoments);

                #endregion
            }
            else
            {
                def_tot = Vector<double>.Build.Dense(1);
                reactions = def_tot;

                internalStresses = Vector<double>.Build.Dense(1);
                internalStrains = internalStresses;
            }

            DA.SetDataList(0, def_tot);
            DA.SetDataList(1, reactions);
            DA.SetDataList(2, internalStresses);
            DA.SetDataList(3, internalStrains);
        }

        private void CalculateInternalStrainsAndStresses(Vector<double>
            def, List<Point3d> vertices, List<MeshFace> faces,
            Matrix<double> B, List<int> BOrder, List<Point3d>
            uniqueNodes, List<Line> edges, double E, double t, double nu,
            out Vector<double> internalStresses, out Vector<double>
            internalStrains, out Vector<double> MorleyMoments)
        {
            //preallocating lists
            internalStresses = Vector<double>.Build.Dense(faces.Count*6);
            internalStrains = Vector<double>.Build.Dense(faces.Count*6);
            MorleyMoments = Vector<double>.Build.Dense(faces.Count*3);
            Matrix <double> C = Matrix<double>.Build.Dense(3, 3);
            C[0, 0] = 1;
            C[0, 1] = nu;
```

```
            C[1, 0] = nu;
            C[1, 1] = 1;
            C[2, 2] = (1 - nu) * 0.5;
            double C_add = E / (1 - Math.Pow(nu, 2));
            C = C_add * C;

            for (int i = 0; i < faces.Count; i++)
            {
                #region Get necessary coordinates and indices
                int indexA = uniqueNodes.IndexOf(vertices[faces[i].A]);
                int indexB = uniqueNodes.IndexOf(vertices[faces[i].B]);
                int indexC = uniqueNodes.IndexOf(vertices[faces[i].C]);

                Point3d verticeA = uniqueNodes[indexA];
                Point3d verticeB = uniqueNodes[indexB];
                Point3d verticeC = uniqueNodes[indexC];

                int edgeIndex1 = edges.IndexOf(new Line(verticeA,
                    verticeB));
                if (edgeIndex1 == -1) { edgeIndex1 = edges.IndexOf(new
                    Line(verticeB, verticeA)); }
                int edgeIndex2 = edges.IndexOf(new Line(verticeB,
                    verticeC));
                if (edgeIndex2 == -1) { edgeIndex2 = edges.IndexOf(new
                    Line(verticeC, verticeB)); }
                int edgeIndex3 = edges.IndexOf(new Line(verticeC,
                    verticeA));
                if (edgeIndex3 == -1) { edgeIndex3 = edges.IndexOf(new
                    Line(verticeA, verticeC)); }

                double x1 = verticeA.X;
                double x2 = verticeB.X;
                double x3 = verticeC.X;

                double y1 = verticeA.Y;
                double y2 = verticeB.Y;
                double y3 = verticeC.Y;

                double z1 = verticeA.Z;
                double z2 = verticeB.Z;
                double z3 = verticeC.Z;
                #endregion

                #region Find tranformation matrix
```

```
            // determine direction cosines for tranformation matrix
            double Lx = Math.Sqrt((Math.Pow((x1 - x2), 2) +
                Math.Pow((y1 - y2), 2) + Math.Pow((z1 - z2), 2)));
            double cosxX = -(x1 - x2) / Lx;
            double cosxY = -(y1 - y2) / Lx;
            double cosxZ = -(z1 - z2) / Lx;
            double Ly = Math.Sqrt((Math.Pow(((y1 - y2) * ((x1 - x2) *
                (y1 - y3) - (x1 - x3) * (y1 - y2)) + (z1 - z2) * ((x1
                - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2))), 2) +
                Math.Pow(((x1 - x2) * ((x1 - x2) * (y1 - y3) - (x1 -
                x3) * (y1 - y2)) - (z1 - z2) * ((y1 - y2) * (z1 - z3)
                - (y1 - y3) * (z1 - z2))), 2) + Math.Pow(((x1 - x2) *
                ((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2)) + (y1
                - y2) * ((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 -
                z2))), 2)));
            double cosyX = ((y1 - y2) * ((x1 - x2) * (y1 - y3) - (x1
                - x3) * (y1 - y2)) + (z1 - z2) * ((x1 - x2) * (z1 -
                z3) - (x1 - x3) * (z1 - z2))) / Ly;
            double cosyY = -((x1 - x2) * ((x1 - x2) * (y1 - y3) - (x1
                - x3) * (y1 - y2)) - (z1 - z2) * ((y1 - y2) * (z1 -
                z3) - (y1 - y3) * (z1 - z2))) / Ly;
            double cosyZ = -((x1 - x2) * ((x1 - x2) * (z1 - z3) - (x1
                - x3) * (z1 - z2)) + (y1 - y2) * ((y1 - y2) * (z1 -
                z3) - (y1 - y3) * (z1 - z2))) / Ly;
            double Lz = Math.Sqrt((Math.Pow(((x1 - x2) * (y1 - y3) -
                (x1 - x3) * (y1 - y2)), 2) + Math.Pow(((x1 - x2) *
                (z1 - z3) - (x1 - x3) * (z1 - z2)), 2) +
                Math.Pow(((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 -
                z2)), 2)));
            double coszX = ((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 -
                z2)) / Lz;
            double coszY = -((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1
                - z2)) / Lz;
            double coszZ = ((x1 - x2) * (y1 - y3) - (x1 - x3) * (y1 -
                y2)) / Lz;

            // assembling nodal x,y,z tranformation matrix tf
            Matrix<double> tf = Matrix<double>.Build.Dense(3, 3);
            tf[0, 0] = cosxX;
            tf[0, 1] = cosxY;
            tf[0, 2] = cosxZ;
            tf[1, 0] = cosyX;
            tf[1, 1] = cosyY;
```

```
tf[1, 2] = cosyZ;
tf[2, 0] = coszX;
tf[2, 1] = coszY;
tf[2, 2] = coszZ;

Matrix<double> T = tf.DiagonalStack(tf);
T = T.DiagonalStack(tf);
Matrix<double> one = Matrix<double>.Build.DenseIdentity(3, 3);
T = T.DiagonalStack(one); // rotations are not transformed
Matrix<double> T_T = T.Transpose();
#endregion

#region Extract B matrices from CST and Morley
Matrix< double> CSTB = Matrix<double>.Build.Dense(3, 6);
Matrix<double> MorleyB = Matrix<double>.Build.Dense(3, 6);
for (int row = 0; row < 3; row++)
{
    for (int col = 0; col < 6; col++)
    {
        CSTB[row, col] = B[row + 6 * i, col];
        MorleyB[row, col] = B[row + 3 + 6 * i, col];
    }
}
//CSTB = B.SubMatrix(6 * i, 3, 0, 6);
//CSTB = B.SubMatrix(6 * i + 3, 3, 0, 6);
#endregion

#region Extract displacement/rotations corresponding to B matrices
Vector<double> CSTv = Vector<double>.Build.Dense(6);
Vector<double> Morleyv = Vector<double>.Build.Dense(6);
for (int j = 0; j < 6; j++)
{
    CSTv[j] = def[BOrder[i * 12 + j]];
    Morleyv[j] = def[BOrder[i * 12 + 6 + j]];
}
#endregion

#region Sort the displacements/rotations to use the tranformation matrix
Vector<double> v = Vector<double>.Build.Dense(12);
int cstc = 0;
int morleyc = 0;
```

```
        for (int k = 0; k < 11; k++)
        {
            if (k < 9)
            {
                if (k == 2 || k == 5 || k == 8)
                {
                    v[k] = Morleyv[morleyc];
                    morleyc++;
                }
                else
                {
                    v[k] = CSTv[cstc];
                    cstc++;
                }
            }
            else
            {
                v[k] = Morleyv[morleyc];
                morleyc++;
            }
        }
        #endregion

        // Transform global deformations to local deformations
        Vector<double> vlocal = T_T.Multiply(v);

        #region Sort the (now local) dofs vlocal and separate CST
            and Morley dofs
        cstc = 0;
        morleyc = 0;
        for (int k = 0; k < 11; k++)
        {
            if (k==2 || k==5 || k==8 || k > 8)
            {
                Morleyv[morleyc] = vlocal[k];
                morleyc++;
            }
            else
            {
                CSTv[cstc] = vlocal[k];
                cstc++;
            }
        }
        #endregion
```

```

            // Calculate CST strain and stress
            Vector<double> CSTstrains = CSTB.Multiply(CSTv);
            Vector<double> CSTstress = C.Multiply(CSTstrains);

            // Calculate Morley strain and stress
            Vector<double> Morleystrains = -t * 0.5 *
                (MorleyB.Multiply(Morleyv));
            Vector<double> Morleystress = C.Multiply(Morleystrains);
            Vector<double> MorleyMoment = t * t / 6.0 *
                C.Multiply(Morleystrains);

            for (int j = 0; j < 3; j++)
            {
                internalStrains[i * 6 + j] = CSTstrains[j];
                internalStrains[i * 6 + 3 + j] = Morleystrains[j];
                internalStresses[i * 6 + j] = CSTstress[j];
                internalStresses[i * 6 + 3 + j] = Morleystress[j];
                MorleyMoments[i * 3 + j] = MorleyMoment[j];
            }
        }
    }

    private Vector<double>
        RestoreTotalDeformationVector(Vector<double>
        deformations_red, Vector<double> bdc_value, Vector<double>
        nakededges)
    {
        Vector<double> def =
            Vector<double>.Build.Dense(bdc_value.Count);
        for (int i = 0, j = 0; i < bdc_value.Count; i++)
        {
            if (bdc_value[i] == 1)
            {
                def[i] = deformations_red[j];
                j++;
            }
        }
        return def;
    }

    private void CreateReducedGlobalStiffnessMatrix(Vector<double>
        bdc_value, Matrix<double> K, List<double> load, List<Point3d>
        uniqueNodes, Vector<double> nakededges, out Matrix<double>
```

```
        K_red, out Vector<double> load_red)
    {
        List<string> placements = new List<string>();
        int oldRC = load.Count;
        int newRC = Convert.ToInt16(bdc_value.Sum());
        K_red = Matrix<double>.Build.Dense(newRC, newRC, 0);
        load_red = Vector<double>.Build.Dense(newRC, 0);
        double K_temp = 0;
        for (int i = 0, ii = 0; i < oldRC; i++)
        {
            //is bdc_value in row i free?
            if (bdc_value[i] == 1)
            {
                for (int j = 0, jj = 0; j <= i; j++)
                {
                    //is bdc_value in col j free?
                    if (bdc_value[j] == 1)
                    {
                        //if yes, then add to new K
                        K_temp = K[i, j];
                        K_red[i - ii, j - jj] = K_temp;
                        K_red[j - jj, i - ii] = K_temp;
                    }
                    else
                    {
                        jj++;
                    }
                }
                //add to reduced load list
                load_red[i - ii] = load[i];
            }
            else
            {
                ii++;
            }
        }
    }

    private void GetUniqueNodes(List<Point3d> vertices, out
        List<Point3d> uniqueNodes)
    {
        uniqueNodes = new List<Point3d>();
        for (int i = 0; i < vertices.Count; i++)
        {
```

```
                Point3d tempNode = new Point3d(Math.Round(vertices[i].X,
                    4), Math.Round(vertices[i].Y, 4),
                    Math.Round(vertices[i].Z, 4));
                if (!uniqueNodes.Contains(tempNode))
                {
                    uniqueNodes.Add(tempNode);
                }
            }
        }

        private void GlobalStiffnessMatrix(List<MeshFace> faces,
            List<Point3d> vertices, List<Line> edges, List<Point3d>
            uniqueNodes, int gdofs, double E, double G, double nu, double
            t, out Matrix<double> KG, out Matrix<double> B, out List<int>
            BDefOrder)
        {
            int NoOfFaces = faces.Count;
            int nodeDofs = uniqueNodes.Count * 3;

            // Want to keep the B matrices for later calculations, we
                also should keep the indices for nodes and edges for speed
            B = Matrix<double>.Build.Dense(NoOfFaces * 6, 6);
            BDefOrder = new List<int>(NoOfFaces * 6);
            int Bcount = 0;

            KG = Matrix<double>.Build.Dense(gdofs, gdofs);

            foreach (var face in faces)
            {
                int indexA = uniqueNodes.IndexOf(vertices[face.A]);
                int indexB = uniqueNodes.IndexOf(vertices[face.B]);
                int indexC = uniqueNodes.IndexOf(vertices[face.C]);

                Point3d verticeA = uniqueNodes[indexA];
                Point3d verticeB = uniqueNodes[indexB];
                Point3d verticeC = uniqueNodes[indexC];

                int edgeIndex1 = edges.IndexOf(new Line(verticeA,
                    verticeB));
                if (edgeIndex1 == -1) { edgeIndex1 = edges.IndexOf(new
                    Line(verticeB, verticeA)); }
                int edgeIndex2 = edges.IndexOf(new Line(verticeB,
                    verticeC));
                if (edgeIndex2 == -1) { edgeIndex2 = edges.IndexOf(new
```

```
    Line(verticeC, verticeB)); }
int edgeIndex3 = edges.IndexOf(new Line(verticeC, verticeA));
if (edgeIndex3 == -1) { edgeIndex3 = edges.IndexOf(new Line(verticeA, verticeC)); }

int[] eindx = new int[] { edgeIndex1, edgeIndex2, edgeIndex3 };
int[] vindx = new int[] { indexA, indexB, indexC };

double x1 = verticeA.X;
double x2 = verticeB.X;
double x3 = verticeC.X;

double y1 = verticeA.Y;
double y2 = verticeB.Y;
double y3 = verticeC.Y;

double z1 = verticeA.Z;
double z2 = verticeB.Z;
double z3 = verticeC.Z;

double[] xList = new double[3] { x1, x2, x3 };
double[] yList = new double[3] { y1, y2, y3 };
double[] zList = new double[3] { z1, z2, z3 };

Matrix<double> Ke; // given as [x1 y1 z1 phi1 x2 y2 z2 phi2 x3 y3 z3 phi3]
Matrix<double> Be;
ElementStiffnessMatrix(xList, yList, zList, E, nu, t, out Ke, out Be);
for (int r = 0; r < 6; r++)
{
    for (int c = 0; c < 6; c++)
    {
        B[Bcount * 6 + r, c] = Be[r, c];
    }
}
//B.SetSubMatrix(Bcount * 6, 0, Be);
Bcount++;
BDefOrder.AddRange(new int[] { indexA * 3, indexA * 3 + 1, indexB * 3, indexB * 3 + 1, indexC * 3, indexC * 3 + 1, indexA * 3 + 2, indexB * 3 + 2, indexC * 3 + 2, nodeDofs + eindx[0], nodeDofs + eindx[1], nodeDofs +
```

```
                eindx[2] });

        for (int row = 0; row < 3; row++)
        {
            for (int col = 0; col < 3; col++)
            {
                //top left 3x3 of K-element matrix
                KG[indexA * 3 + row, indexA * 3 + col] += Ke[row,
                    col];
                //top middle 3x3 of k-element matrix
                KG[indexA * 3 + row, indexB * 3 + col] += Ke[row,
                    col + 4];
                //top right 3x3 of k-element matrix
                KG[indexA * 3 + row, indexC * 3 + col] += Ke[row,
                    col + 4 * 2];

                //middle left 3x3 of k-element matrix
                KG[indexB * 3 + row, indexA * 3 + col] += Ke[row
                    + 4, col];
                //middle middle 3x3 of k-element matrix
                KG[indexB * 3 + row, indexB * 3 + col] += Ke[row
                    + 4, col + 4];
                //middle right 3x3 of k-element matrix
                KG[indexB * 3 + row, indexC * 3 + col] += Ke[row
                    + 4, col + 4 * 2];

                //bottom left 3x3 of k-element matrix
                KG[indexC * 3 + row, indexA * 3 + col] += Ke[row
                    + 4 * 2, col];
                //bottom middle 3x3 of k-element matrix
                KG[indexC * 3 + row, indexB * 3 + col] += Ke[row
                    + 4 * 2, col + 4];
                //bottom right 3x3 of k-element matrix
                KG[indexC * 3 + row, indexC * 3 + col] += Ke[row
                    + 4 * 2, col + 4 * 2];

                // insert rotations for edges in correct place
                //Rotation to rotation relation
                KG[nodeDofs + eindx[row], nodeDofs + eindx[col]]
                    += Ke[row * 4 + 3, col * 4 + 3];

                //Rotation to x relation lower left
                KG[nodeDofs + eindx[row], vindx[col] * 3] +=
                    Ke[row * 4 + 3, col * 4];
```

```
                        //Rotation to x relation upper right
                        KG[vindx[row] * 3, nodeDofs + eindx[col]] += Ke[row * 4, col * 4 + 3];

                        //Rotation to y relation lower left
                        KG[nodeDofs + eindx[row], vindx[col] * 3 + 1] += Ke[row * 4 + 3, col * 4 + 1];
                        //Rotation to y relation upper right
                        KG[vindx[row] * 3 + 1, nodeDofs + eindx[col]] += Ke[row * 4 + 1, col * 4 + 3];

                        //Rotation to z relation lower left
                        KG[nodeDofs + eindx[row], vindx[col] * 3 + 2] += Ke[row * 4 + 3, col * 4 + 2];
                        //Rotation to z relation upper right
                        KG[vindx[row] * 3 + 2, nodeDofs + eindx[col]] += Ke[row * 4 + 2, col * 4 + 3];
                    }
                }
            }
        }

        private void ElementStiffnessMatrix(double[] xList, double[] yList, double[] zList, double E, double nu, double t, out Matrix<double> Ke, out Matrix<double> B)
        {

            #region Get global coordinates and transform into local cartesian system

            // fetching global coordinates
            double x1 = xList[0];
            double x2 = xList[1];
            double x3 = xList[2];

            double y1 = yList[0];
            double y2 = yList[1];
            double y3 = yList[2];

            double z1 = zList[0];
            double z2 = zList[1];
            double z3 = zList[2];

            // determine angles for tranformation matrix
```

```
            double Lx = Math.Sqrt((Math.Pow((x1 - x2), 2) + Math.Pow((y1 - y2), 2) + Math.Pow((z1 - z2), 2)));
            double cosxX = -(x1 - x2) / Lx;
            double cosxY = -(y1 - y2) / Lx;
            double cosxZ = -(z1 - z2) / Lx;
            double Ly = Math.Sqrt((Math.Pow(((y1 - y2) * ((x1 - x2) * (y1 - y3) - (x1 - x3) * (y1 - y2)) + (z1 - z2) * ((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2))), 2) + Math.Pow(((x1 - x2) * ((x1 - x2) * (y1 - y3) - (x1 - x3) * (y1 - y2)) - (z1 - z2) * ((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 - z2))), 2) + Math.Pow(((x1 - x2) * ((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2)) + (y1 - y2) * ((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 - z2))), 2)));
            double cosyX = ((y1 - y2) * ((x1 - x2) * (y1 - y3) - (x1 - x3) * (y1 - y2)) + (z1 - z2) * ((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2))) / Ly;
            double cosyY = -((x1 - x2) * ((x1 - x2) * (y1 - y3) - (x1 - x3) * (y1 - y2)) - (z1 - z2) * ((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 - z2))) / Ly;
            double cosyZ = -((x1 - x2) * ((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2)) + (y1 - y2) * ((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 - z2))) / Ly;
            double Lz = Math.Sqrt((Math.Pow(((x1 - x2) * (y1 - y3) - (x1 - x3) * (y1 - y2)), 2) + Math.Pow(((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2)), 2) + Math.Pow(((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 - z2)), 2)));
            double coszX = ((y1 - y2) * (z1 - z3) - (y1 - y3) * (z1 - z2)) / Lz;
            double coszY = -((x1 - x2) * (z1 - z3) - (x1 - x3) * (z1 - z2)) / Lz;
            double coszZ = ((x1 - x2) * (y1 - y3) - (x1 - x3) * (y1 - y2)) / Lz;

            // assembling nodal x,y,z tranformation matrix tf
            Matrix<double> tf = Matrix<double>.Build.Dense(3, 3);
            tf[0, 0] = cosxX;
            tf[0, 1] = cosxY;
            tf[0, 2] = cosxZ;
            tf[1, 0] = cosyX;
            tf[1, 1] = cosyY;
            tf[1, 2] = cosyZ;
            tf[2, 0] = coszX;
            tf[2, 1] = coszY;
            tf[2, 2] = coszZ;
```

```
        // assemble the full transformation matrix T for the entire
            element (12x12 matrix)
        Matrix<double> one = Matrix<double>.Build.Dense(1, 1, 1);
        var T = tf;
        T = T.DiagonalStack(one);
        T = T.DiagonalStack(tf);
        T = T.DiagonalStack(one);
        T = T.DiagonalStack(tf);
        T = T.DiagonalStack(one);
        Matrix<double> T_T = T.Transpose(); // and the transposed
            tranformation matrix

        // initiates the local coordinate matrix, initiated with
            global coordinates
        Matrix<double> lcoord = Matrix<double>.Build.DenseOfArray(new
            double[,]
        {
            { x1, x2, x3 },
            { y1, y2, y3 },
            { z1, z2, z3 }
        });

        //transforms lcoord into local coordinate values
        lcoord = tf.Multiply(lcoord);

        // sets the new (local) coordinate values
        x1 = lcoord[0, 0];
        x2 = lcoord[0, 1];
        x3 = lcoord[0, 2];
        y1 = lcoord[1, 0];
        y2 = lcoord[1, 1];
        y3 = lcoord[1, 2];
        z1 = lcoord[2, 0];
        z2 = lcoord[2, 1];
        z3 = lcoord[2, 2]; // Note that z1 = z2 = z3, if all goes
            according to plan

        #endregion

        double Area = Math.Abs(0.5 * (x1 * (y2 - y3) + x2 * (y3 - y1)
            + x3 * (y1 - y2)));

        // Establishes the general flexural rigidity matrix for plate
```

```
            Matrix<double> C = Matrix<double>.Build.Dense(3, 3);
            C[0, 0] = 1;
            C[0, 1] = nu;
            C[1, 0] = nu;
            C[1, 1] = 1;
            C[2, 2] = (1 - nu)*0.5;

            double C_add = E / (1 - Math.Pow(nu, 2)); // additional part
                to add to every indice in C matrix

            #region Morley Bending Triangle -- Bending part of element
                gives [z1 z2 z3 phi1 phi2 phi3]

            Matrix<double> lcoord_temp =
                Matrix<double>.Build.DenseOfArray(new double[,] { { x1 },
                { y1 }, { z1 } });
            lcoord = lcoord.Append(lcoord_temp);

            // defines variables for simplicity
            double x13 = x1 - x3;
            double x32 = x3 - x2;
            double y23 = y2 - y3;
            double y31 = y3 - y1;

            double[] ga = new double[3];
            double[] my = new double[3];
            double[] a = new double[3];

            for (int i = 0; i < 3; i++)
            {
                double c, s;
                double len = Math.Sqrt(Math.Pow(lcoord[0, i + 1] -
                    lcoord[0, i], 2) + Math.Pow(lcoord[1, i + 1] -
                    lcoord[1, i], 2));
                if (lcoord[0, i + 1] > lcoord[0, i])
                {
                    c = (lcoord[0, i + 1] - lcoord[0, i]) / len;
                    s = (lcoord[1, i + 1] - lcoord[1, i]) / len;
                }
                else if (lcoord[0, i + 1] < lcoord[0, i])
                {
                    c = (lcoord[0, i] - lcoord[0, i + 1]) / len;
                    s = (lcoord[1, i] - lcoord[1, i + 1]) / len;
                }
```

```
            else
            {
                c = 0.0;
                s = 1.0;
            }
            ga[i] = (c * x32 - s * y23) / (2 * Area);
            my[i] = (c * x13 - s * y31) / (2 * Area);
            a[i] = ga[i] + my[i];
        }

        double ga4 = ga[0];
        double ga5 = ga[1];
        double ga6 = ga[2];
        double my4 = my[0];
        double my5 = my[1];
        double my6 = my[2];
        double a4 = a[0];
        double a5 = a[1];
        double a6 = a[2];

        Matrix<double> Bk_b = Matrix<double>.Build.Dense(3, 6); //
            Exported from Matlab
        Bk_b[0, 0] = -(2 * (ga4 * my6 * Math.Pow(y23, 2) - a4 * my6 *
            Math.Pow(y23, 2) - a4 * ga6 * Math.Pow(y31, 2) + ga4 *
            my6 * Math.Pow(y31, 2) + 2 * ga4 * my6 * y23 * y31)) /
            (a4 * my6);
        Bk_b[0, 1] = -(2 * (ga5 * my4 * Math.Pow(y23, 2) - a4 * my5 *
            Math.Pow(y23, 2) - a4 * ga5 * Math.Pow(y31, 2) + ga5 *
            my4 * Math.Pow(y31, 2) + 2 * ga5 * my4 * y23 * y31)) /
            (a4 * ga5);
        Bk_b[0, 2] = (2 * (ga5 * my6 * Math.Pow(y23, 2) - a5 * my6 *
            Math.Pow(y23, 2) - a6 * ga5 * Math.Pow(y31, 2) + ga5 *
            my6 * Math.Pow(y31, 2) + 2 * ga5 * my6 * y23 * y31)) /
            (ga5 * my6);
        Bk_b[0, 3] = (2 * Math.Pow((y23 + y31), 2)) / a4;
        Bk_b[0, 4] = -(2 * Math.Pow(y23, 2)) / ga5;
        Bk_b[0, 5] = -(2 * Math.Pow(y31, 2)) / my6;
        Bk_b[1, 0] = -(2 * (ga4 * my6 * Math.Pow(x13, 2) - a4 * my6 *
            Math.Pow(x32, 2) + ga4 * my6 * Math.Pow(x32, 2) - a4 *
            ga6 * Math.Pow(x13, 2) + 2 * ga4 * my6 * x13 * x32)) /
            (a4 * my6);
        Bk_b[1, 1] = -(2 * (ga5 * my4 * Math.Pow(x13, 2) - a4 * my5 *
            Math.Pow(x32, 2) + ga5 * my4 * Math.Pow(x32, 2) - a4 *
            ga5 * Math.Pow(x13, 2) + 2 * ga5 * my4 * x13 * x32)) /
```

```
    (a4 * ga5);
            Bk_b[1, 2] = (2 * (ga5 * my6 * Math.Pow(x13, 2) - a5 * my6 * Math.Pow(x32, 2) + ga5 * my6 * Math.Pow(x32, 2) - a6 * ga5 * Math.Pow(x13, 2) + 2 * ga5 * my6 * x13 * x32)) / (ga5 * my6);
            Bk_b[1, 3] = (2 * Math.Pow((x13 + x32), 2)) / a4;
            Bk_b[1, 4] = -(2 * Math.Pow(x32, 2)) / ga5;
            Bk_b[1, 5] = -(2 * Math.Pow(x13, 2)) / my6;
            Bk_b[2, 0] = -(4 * (ga4 * my6 * x13 * y23 - a4 * my6 * x32 * y23 - a4 * ga6 * x13 * y31 + ga4 * my6 * x13 * y31 + ga4 * my6 * x32 * y23 + ga4 * my6 * x32 * y31)) / (a4 * my6);
            Bk_b[2, 1] = -(4 * (ga5 * my4 * x13 * y23 - a4 * my5 * x32 * y23 - a4 * ga5 * x13 * y31 + ga5 * my4 * x13 * y31 + ga5 * my4 * x32 * y23 + ga5 * my4 * x32 * y31)) / (a4 * ga5);
            Bk_b[2, 2] = 4 * x13 * y23 + 4 * x13 * y31 + 4 * x32 * y23 + 4 * x32 * y31 - (4 * a5 * x32 * y23) / ga5 - (4 * a6 * x13 * y31) / my6;
            Bk_b[2, 3] = (4 * (x13 + x32) * (y23 + y31)) / a4;
            Bk_b[2, 4] = -(4 * x32 * y23) / ga5;
            Bk_b[2, 5] = -(4 * x13 * y31) / my6;

            double Bk_b_add = 1 / (4.0 * Math.Pow(Area, 2)); // additional part to add to every indice in B matrix

            Matrix<double> Bk_b_T = Bk_b.Transpose();

            Matrix<double> ke_b = C.Multiply(Bk_b); // the bending part of the element stiffness matrix
            ke_b = Bk_b_T.Multiply(ke_b);
            double ke_b_add = (Area * t * t * t) / 12; // additional part to add to every indice in ke_b matrix
            ke_b_add = ke_b_add * Bk_b_add * C_add * Bk_b_add; // multiply upp all additional parts
            ke_b = ke_b.Multiply(ke_b_add);

#endregion


            #region Constant Strain/Stress Triangle (CST) -- Membrane part of element gives [x1 y1 x2 y2 x3 y3]

            Matrix<double> Bk_m = Matrix<double>.Build.Dense(3, 6); // Exported from Matlab

```

```
            Bk_m[0, 0] = (y2 - y3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[0, 2] = -(y1 - y3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[0, 4] = (y1 - y2) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[1, 1] = -(x2 - x3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[1, 3] = (x1 - x3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[1, 5] = -(x1 - x2) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[2, 0] = -(x2 - x3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[2, 1] = (y2 - y3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[2, 2] = (x1 - x3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[2, 3] = -(y1 - y3) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[2, 4] = -(x1 - x2) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);
            Bk_m[2, 5] = (y1 - y2) / (x1 * y2 - x2 * y1 - x1 * y3 + x3 * y1 + x2 * y3 - x3 * y2);

            Matrix<double> Bk_m_T = Bk_m.Transpose();

            Matrix<double> ke_m = C.Multiply(Bk_m); // the membrane part of the element stiffness matrix
            ke_m = Bk_m_T.Multiply(ke_m);
            ke_m = ke_m.Multiply(C_add * Area * t);

            #endregion

            B = Bk_m.Stack(Bk_b*Bk_b_add);

            // input membrane and bending part into full element stiffness matrix
            // and stacking them from [x1 y1 x2 y2 x3 y3 z1 z2 z3 phi1 phi2 phi3]
            // into [x1 y1 z1 phi1 x2 y2 z2 phi2 x3 y3 z3 phi3] which gives the stacking order: { 0 1 6 9 2 3 7 10 4 5 8 11 }
            Matrix<double> ke = ke_m.DiagonalStack(ke_b);
            ke = SymmetricRearrangeMatrix(ke, new int[] { 0, 1, 6, 9, 2,
```

```
            3, 7, 10, 4, 5, 8, 11 }, 12); //strictly not necessary, but is done for simplicity and understandability


        Ke = ke.Multiply(T);
        Ke = T_T.Multiply(Ke);
    }

    private Matrix<double> RearrangeMatrixRows(Matrix<double> M, int[] arrangement, int row, int col)
    {
        Matrix<double> M_new = Matrix<double>.Build.Dense(row, col);

        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < col; j++)
            {
                M_new[i, j] = M[arrangement[i],j];
            }
        }
        return M_new;
    }

    private Matrix<double> SymmetricRearrangeMatrix(Matrix<double> M, int[] arrangement, int rowcol)
    {
        Matrix<double> M_new =
            Matrix<double>.Build.Dense(rowcol,rowcol);

        for (int i = 0; i < rowcol; i++)
        {
            for (int j = 0; j < rowcol; j++)
            {
                M_new[i, j] = M[arrangement[i], arrangement[j]];
            }
        }
        return M_new;
    }

    private List<double> CreateLoadList(List<string> loadtxt, List<string> momenttxt, List<Point3d> uniqueNodes, List<MeshFace> faces, List<Point3d> vertices, List<Line> edges)
    {
```

```
            //initializing loads with list of doubles of size gdofs and
                entry values = 0
            List<double> loads = new List<double>(new
                double[uniqueNodes.Count * 3 + edges.Count]);
            List<double> inputLoads = new List<double>();
            List<Point3d> coordlist = new List<Point3d>();

            //parsing point loads
            for (int i = 0; i < loadtxt.Count; i++)
            {
                string coordstr = (loadtxt[i].Split(':')[0]);
                string loadstr = (loadtxt[i].Split(':')[1]);

                string[] coordstr1 = (coordstr.Split(','));
                string[] loadstr1 = (loadstr.Split(','));

                inputLoads.Add(Math.Round(double.Parse(loadstr1[0]), 2));
                inputLoads.Add(Math.Round(double.Parse(loadstr1[1]), 2));
                inputLoads.Add(Math.Round(double.Parse(loadstr1[2]), 2));

                coordlist.Add(new
                    Point3d(Math.Round(double.Parse(coordstr1[0]), 4),
                    Math.Round(double.Parse(coordstr1[1]), 4),
                    Math.Round(double.Parse(coordstr1[2]), 4)));
            }

            //inputting point loads at correct index in loads list
            foreach (Point3d point in coordlist)
            {
                int gNodeIndex = uniqueNodes.IndexOf(point);
                int lNodeIndex = coordlist.IndexOf(point);
                loads[gNodeIndex * 3 + 0] = inputLoads[lNodeIndex * 3 +
                    0];
                loads[gNodeIndex * 3 + 1] = inputLoads[lNodeIndex * 3 +
                    1];
                loads[gNodeIndex * 3 + 2] = inputLoads[lNodeIndex * 3 +
                    2];
            }
            //resetting variables
            inputLoads.Clear();
            coordlist.Clear();

            return loads;
        }
```

```
        private Vector<double> CreateBDCList(List<string> bdctxt,
            List<Point3d> uniqueNodes, List<MeshFace> faces,
            List<Point3d> vertices, List<Line> edges)
        {
            //initializing bdc_value as vector of size gdofs, and entry
                values = 1
            Vector<double> bdc_value =
                Vector.Build.Dense(uniqueNodes.Count * 3 + edges.Count
                ,1);
            List<int> bdcs = new List<int>();
            List<Point3d> bdc_points = new List<Point3d>(); //Coordinates
                relating til bdc_value in for (eg. x y z)
            List<int> fixedRotEdges = new List<int>();
            int rows = bdctxt.Count;

            //Parse string input
            int numOfPoints = bdctxt.Count;
            for (int i = 0; i < numOfPoints; i++)
            {
                if (bdctxt[i] == null)
                {
                    continue;
                }
                else if (!bdctxt[i].Contains(":"))
                {
                    string[] edgestrtemp = bdctxt[i].Split(',');
                    List<string> edgestr = new List<string>();
                    edgestr.AddRange(edgestrtemp);
                    for (int j = 0; j < edgestr.Count; j++)
                    {
                        fixedRotEdges.Add(int.Parse(edgestr[j]));
                    }
                    continue;
                }
                string coordstr = bdctxt[i].Split(':')[0];
                string bdcstr = bdctxt[i].Split(':')[1];

                string[] coordstr1 = (coordstr.Split(','));
                string[] bdcstr1 = (bdcstr.Split(','));

                bdc_points.Add(new
                    Point3d(Math.Round(double.Parse(coordstr1[0]), 4),
                    Math.Round(double.Parse(coordstr1[1]), 4),
```

```
                    Math.Round(double.Parse(coordstr1[2]), 4)));

                bdcs.Add(int.Parse(bdcstr1[0]));
                bdcs.Add(int.Parse(bdcstr1[1]));
                bdcs.Add(int.Parse(bdcstr1[2]));
            }

            //Format to correct entries in bdc_value

            foreach (var point in bdc_points)
            {
                int index = bdc_points.IndexOf(point);
                int i = uniqueNodes.IndexOf(point);
                bdc_value[i * 3 + 0] = bdcs[index * 3 + 0];
                bdc_value[i * 3 + 1] = bdcs[index * 3 + 1];
                bdc_value[i * 3 + 2] = bdcs[index * 3 + 2];
            }

            foreach (var edgeindex in fixedRotEdges)
            {
                bdc_value[edgeindex+uniqueNodes.Count*3] = 0;
            }


            return bdc_value;
        }

        private void SetMaterial(string mattxt, out double E, out double
            G, out double nu, out double t)
        {
            string[] matProp = (mattxt.Split(','));
            E = (Math.Round(double.Parse(matProp[0]), 2));
            nu = (Math.Round(double.Parse(matProp[1]), 3));
            t = (Math.Round(double.Parse(matProp[2]), 2));
            if (matProp.GetLength(0) == 4)
            {
                G = (Math.Round(double.Parse(matProp[3]), 2));
            }
            else
            {
                G = E / (2.0 * (1.0 + nu));
            }

        }
```

```
        protected override System.Drawing.Bitmap Icon
        {
            get
            {

                return Properties.Resources.Calc1;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("3a61d696-911f-46cd-a687-ef48a48575b0"); }
        }

        /// Component Visual//
        public class Attributes_Custom :
            Grasshopper.Kernel.Attributes.GH_ComponentAttributes
        {
            public Attributes_Custom(GH_Component owner) : base(owner) { }
            protected override void Layout()
            {
                base.Layout();

                Rectangle rec0 = GH_Convert.ToRectangle(Bounds);

                rec0.Height += 22;

                Rectangle rec1 = rec0;
                rec1.X = rec0.Left + 1;
                rec1.Y = rec0.Bottom - 22;
                rec1.Width = (rec0.Width) / 3 + 1;
                rec1.Height = 22;
                rec1.Inflate(-2, -2);

                Rectangle rec2 = rec1;
                rec2.X = rec1.Right + 2;

                Bounds = rec0;
                ButtonBounds = rec1;
                ButtonBounds2 = rec2;

            }
```

```
        GH_Palette xColor = GH_Palette.Black;
        GH_Palette yColor = GH_Palette.Grey;

        private Rectangle ButtonBounds { get; set; }
        private Rectangle ButtonBounds2 { get; set; }
        private Rectangle ButtonBounds3 { get; set; }

        protected override void Render(GH_Canvas canvas, Graphics
            graphics, GH_CanvasChannel channel)
        {
            base.Render(canvas, graphics, channel);
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button;
                if (startCalc == true)
                {
                    button =
                        GH_Capsule.CreateTextCapsule(ButtonBounds,
                        ButtonBounds, xColor, "Run: On", 3, 0);
                }
                else
                {
                    button =
                        GH_Capsule.CreateTextCapsule(ButtonBounds,
                        ButtonBounds, yColor, "Run: Off", 3, 0);
                }
                button.Render(graphics, Selected, false, false);
                button.Dispose();
            }
        }

        public override GH_ObjectResponse
            RespondToMouseDown(GH_Canvas sender, GH_CanvasMouseEvent
            e)
        {
            if (e.Button == MouseButtons.Left)
            {
                RectangleF rec = ButtonBounds;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("Run");
                    if (xColor == GH_Palette.Black) { setStart("Run",
                        true); Owner.ExpireSolution(true); }
```

```
                        if (xColor == GH_Palette.Grey) { setStart("Run", false); Owner.ExpireSolution(true); }

                        sender.Refresh();
                        return GH_ObjectResponse.Handled;
                    }
                    rec = ButtonBounds2;
                    if (rec.Contains(e.CanvasLocation))
                    {
                        switchColor("Run Test");
                        if (yColor == GH_Palette.Black) { setStart("Run Test", true); }
                        if (yColor == GH_Palette.Grey) { setStart("Run Test", false); }
                        sender.Refresh();
                        return GH_ObjectResponse.Handled;
                    }
                }
                return base.RespondToMouseDown(sender, e);
            }

            private void switchColor(string button)
            {
                if (button == "Run")
                {
                    if (xColor == GH_Palette.Black) { xColor = GH_Palette.Grey; }
                    else { xColor = GH_Palette.Black; }
                }
                else if (button == "Run Test")
                {
                    if (yColor == GH_Palette.Black) { yColor = GH_Palette.Grey; }
                    else { yColor = GH_Palette.Black; }
                }
            }
        }
    }
}
```

Shell Set Loads Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;

namespace Shell
{
    public class SetLoads : GH_Component
    {
        public SetLoads()
          : base("PointLoads Shell", "PL",
            "Point loads to apply to a shell structure",
            "Koala", "Shell")
        {
        }
        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply load(s)", GH_ParamAccess.list);
            pManager.AddNumberParameter("Load", "L", "Load originally given i Newtons (N), give one load for all points or list of loads for each point", GH_ParamAccess.list);
            pManager.AddNumberParameter("angle (xz)", "axz", "give angle for load in xz plane", GH_ParamAccess.list, 90);
            pManager.AddNumberParameter("angle (xy)", "axy", "give angle for load in xy plane", GH_ParamAccess.list, 0);
            //pManager[2].Optional = true; //Code can run without a given angle (90 degrees is initial value)
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddTextParameter("PointLoads", "PL", "PointLoads formatted for Calculation Component", GH_ParamAccess.list);
        }

```

```
        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch inputs
            //Expected inputs and output
            List<Point3d> pointList = new List<Point3d>();
                //List of points where load will be applied
            List<double> loadList = new List<double>();
                //List or value of load applied
            List<double> anglexz = new List<double>();
                //Initial xz angle 90, angle from x axis in xz plane for
                load
            List<double> anglexy = new List<double>();
                //Initial xy angle 0, angle from x axis in xy plane for
                load
            List<string> pointInStringFormat = new List<string>();
                //preallocate final string output

            //Set expected inputs from Indata
            if (!DA.GetDataList(0, pointList)) return;
            if (!DA.GetDataList(1, loadList)) return;
            if (!DA.GetDataList(2, anglexz)) return;
            if (!DA.GetDataList(3, anglexy)) return;
            #endregion

            #region Format pointloads
            //initialize temporary stringline and load vectors
            string vectorString;
            double load = 0;
            double xvec = 0;
            double yvec = 0;
            double zvec = 0;

            if (loadList.Count == 1 && anglexz.Count == 1)
                //loads and angles are identical for all points
            {
                load = -1 * loadList[0];
                    //negativ load for z-dir
                xvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI /
                    180) * Math.Cos(anglexy[0] * Math.PI / 180), 5);
                yvec = Math.Round(load * Math.Cos(anglexz[0] * Math.PI /
                    180) * Math.Sin(anglexy[0] * Math.PI / 180), 5);
                zvec = Math.Round(load * Math.Sin(anglexz[0] * Math.PI /
                    180), 5);

```

```
                vectorString = xvec + "," + yvec + "," + zvec;
                for (int i = 0; i < pointList.Count; i++)     //adds identical load to all points in pointList
                {
                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + vectorString);
                }
            }
            else   //loads and angles may be different => calculate new xvec, yvec, zvec for all loads
            {
                for (int i = 0; i < pointList.Count; i++)
                {
                    if (loadList.Count < i)             //if pointlist is larger than loadlist, set last load value in remaining points
                    {
                        vectorString = xvec + "," + yvec + "," + zvec;
                    }
                    else
                    {
                        load = -1 * loadList[i];        //negative load for z-dir

                        xvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Cos(anglexy[i]), 2);
                        yvec = Math.Round(load * Math.Cos(anglexz[i]) * Math.Sin(anglexy[i]), 2);
                        zvec = Math.Round(load * Math.Sin(anglexz[i]), 2);

                        vectorString = xvec + "," + yvec + "," + zvec;
                    }

                    pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + vectorString);
                }
            }
            #endregion

            //Set output data
            DA.SetDataList(0, pointInStringFormat);
        }
```

```
        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.Pointloads;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("2935c931-2647-4bc5-b851-68e7d4af9001"); }
        }
    }
}
```

Shell BDC Component

```
using System;
using System.Collections.Generic;

using Grasshopper.Kernel;
using Rhino.Geometry;
using System.Drawing;
using Grasshopper.GUI.Canvas;
using System.Windows.Forms;
using Grasshopper.GUI;

namespace Shell
{
    public class BDCComponent : GH_Component
    {
        public BDCComponent()
          : base("Shell BDC", "BDCs",
              "Description",
              "Koala", "Shell")
        {
        }

        //Initialize BDCs
        static int x = 0;
        static int y = 0;
        static int z = 0;
        static int rx = 0;

        //Method to allow c hanging of variables via GUI (see Component
            Visual)
        public static void setBDC(string s, int i)
        {
            if (s == "X")
            {
                x = i;
            }
            else if (s == "Y")
            {
                y = i;
            }
            else if (s == "Z")
            {
                z = i;
```

```
            }
            else if (s == "RX")
            {
                rx = i;
            }
        }

        public override void CreateAttributes()
        {
            m_attributes = new Attributes_Custom(this);
        }

        protected override void RegisterInputParams(GH_Component.GH_InputParamManager pManager)
        {
            pManager.AddPointParameter("Points", "P", "Points to apply Boundary Conditions", GH_ParamAccess.list);
            pManager.AddMeshParameter("Mesh", "M", "Give mesh if edges should be fixed", GH_ParamAccess.item);
            pManager[1].Optional = true;
        }

        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddTextParameter("B.Cond.", "BDC", "Boundary Conditions for Shell element", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch inputs
            //Expected inputs
            List<Point3d> pointList = new List<Point3d>(); //List of points where BDC is to be applied
            List<string> pointInStringFormat = new List<string>(); //output in form of list of strings

            //Expected inputs
            Mesh mesh = new Mesh();                         //mesh in Mesh format
            List<MeshFace> faces = new List<MeshFace>();    //faces of
```

```
            mesh as a list
        List<Point3d> vertices = new List<Point3d>();   //vertices of mesh as a list

        //Set expected inputs from Indata and aborts with error message if input is incorrect
        if (!DA.GetDataList(0, pointList)) return;
        DA.GetData(1, ref mesh);         //sets inputted mesh into variable
        #endregion

        for (int i = 0; i < pointList.Count; i++)
        {
            Point3d temp_point = new Point3d();
            temp_point.X = Math.Round(pointList[i].X, 4);
            temp_point.Y = Math.Round(pointList[i].Y, 4);
            temp_point.Z = Math.Round(pointList[i].Z, 4);
            pointList[i] = temp_point;
        }


        List<Line> edges = new List<Line>();
        #region If mesh is given and rotations should be fixed
        if (mesh.Faces.Count != 0 && rx == 0)
        {
            foreach (var face in mesh.Faces)
            {
                faces.Add(face);
            }

            foreach (var vertice in mesh.Vertices)
            {
                Point3d temp_vertice = new Point3d();
                temp_vertice.X = Math.Round(vertice.X, 4);
                temp_vertice.Y = Math.Round(vertice.Y, 4);
                temp_vertice.Z = Math.Round(vertice.Z, 4);
                vertices.Add(temp_vertice);
            }
            int NoOfEdges = vertices.Count + faces.Count - 1;
            edges = new List<Line>(NoOfEdges);
            foreach (var face in faces)
            {
                Point3d vA = vertices[face.A];
                Point3d vB = vertices[face.B];
```

```
            Point3d vC = vertices[face.C];
            Line lineAB = new Line(vA, vB);
            Line lineBA = new Line(vB, vA);
            Line lineCB = new Line(vC, vB);
            Line lineBC = new Line(vB, vC);
            Line lineAC = new Line(vA, vC);
            Line lineCA = new Line(vC, vA);

            if (!edges.Contains(lineAB) && !edges.Contains(lineBA))
            {
                edges.Add(lineAB);
            }
            if (!edges.Contains(lineCB) && !edges.Contains(lineBC))
            {
                edges.Add(lineBC);
            }
            if (!edges.Contains(lineAC) && !edges.Contains(lineCA))
            {
                edges.Add(lineAC);
            }
        }
    }
        #endregion

    #region Find edge indexes if fixed rotation and Format output
        string BDCString = x + "," + y + "," + z;


        if (rx == 1 || !mesh.IsValid)
        {
            for (int i = 0; i < pointList.Count; i++)    //Format stringline for all points (identical boundary conditions for all points), no fixed rotations
            {
                pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);
            }
        }
        else
        {
```

```
            int rot = -1;
            List<int> edgeindexrot = new List<int>();
            List<List<int>> mIndices = GetMeshIndices(pointList,
                faces, vertices);
            for (int i = 0; i < pointList.Count; i++)
            {
                if (mIndices.Count == 0) { break; }
                int facenum = -1;
                if (mIndices[i].Count == 1)
                {
                    facenum = mIndices[i][0];
                }
                else if (mIndices[i].Count == 2)
                {
                    facenum = mIndices[i][1];
                }
                else
                {
                    break;
                }
                List<Point3d> connectedPoints = new
                    List<Point3d>();
                for (int j = 0; j < pointList.Count; j++)
                {
                    if (j != i && mIndices[j][0] == facenum)
                    {
                        connectedPoints.Add(pointList[j]);
                    }
                }
                Line bdcline;
                if (connectedPoints.Count >= 1)
                {
                    bdcline = new Line(pointList[i],
                        connectedPoints[0]);
                    if (edges.Contains(bdcline))
                    {
                        rot = edges.IndexOf(bdcline);
                    }
                    bdcline = new Line(connectedPoints[0],
                        pointList[i]);
                    if (edges.Contains(bdcline))
                    {
                        rot = edges.IndexOf(bdcline);
                    }
```

```
        if (!edgeindexrot.Contains(rot) && rot != -1)
        {
            edgeindexrot.Add(rot);
        }
    }

    if (connectedPoints.Count == 2)
    {
        bdcline = new Line(pointList[i], connectedPoints[1]);
        if (edges.Contains(bdcline))
        {
            rot = edges.IndexOf(bdcline);
        }
        bdcline = new Line(connectedPoints[1], pointList[i]);
        if (edges.Contains(bdcline))
        {
            rot = edges.IndexOf(bdcline);
        }
        if (!edgeindexrot.Contains(rot) && rot != -1)
        {
            edgeindexrot.Add(rot);
        }
    }
}

for (int i = 0; i <= pointList.Count; i++)    //Format stringline for all points (identical boundary conditions for all points), no fixed rotations
{
    if (i < pointList.Count)
    {
        pointInStringFormat.Add(pointList[i].X + "," + pointList[i].Y + "," + pointList[i].Z + ":" + BDCString);
    }
    else
    {
        string rotindex = null;
        foreach (var item in edgeindexrot)
        {
            if (item == edgeindexrot[0])
            {
```

```
                                    rotindex += item;
                                }
                                else
                                {
                                    rotindex = rotindex + ',' + item;
                                }
                            }
                            pointInStringFormat.Add(rotindex);
                        }
                    }
                }
                #endregion

            DA.SetDataList(0, pointInStringFormat);
        } //End of main program

        private List<List<int>> GetMeshIndices(List<Point3d> pointList, List<MeshFace> faces, List<Point3d> vertices)
        {
            //initiates list of lists with -1s
            List<List<int>> indices = new List<List<int>>();
            for (int i = 0; i < pointList.Count; i++)
            {
                List<int> tempL = new List<int>();
                //tempL.Add(-1);
                for (int j = 0; j < faces.Count; j++)
                {
                    //is point in mesh?
                    if (pointList[i] == vertices[faces[j].A])
                    {
                        //are any of the other mesh vertices in pointList?
                        if (pointList.Contains(vertices[faces[j].B]) || pointList.Contains(vertices[faces[j].C]))
                        {
                            //indicates that the mesh j contains 2+ vertices and that their edge should be fixed
                            tempL.Add(j);
                            //check if other mesh faces share points (otherwise would have used break;)
                            continue;
                        }
                    }
                    else if (pointList[i] == vertices[faces[j].B])
```

```
                    {
                        if (pointList.Contains(vertices[faces[j].A]) ||
                            pointList.Contains(vertices[faces[j].C]))
                        {
                            tempL.Add(j);
                            continue;
                        }
                    }
                    else if (pointList[i] == vertices[faces[j].C])
                    {
                        if (pointList.Contains(vertices[faces[j].A]) ||
                            pointList.Contains(vertices[faces[j].B]))
                        {
                            tempL.Add(j);
                            continue;
                        }
                    }
                }
                if (tempL.Count > 0)
                {
                    indices.Add(tempL);
                }
                //indices.Add(tempL);
            }
            return indices;
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.BDCs;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("58ccdcb8-b1c3-411b-b501-c91a46665e86"); }
        }

        /// Component Visual//
        public class Attributes_Custom :
            Grasshopper.Kernel.Attributes.GH_ComponentAttributes
```

```
        {
            public Attributes_Custom(GH_Component owner) : base(owner) { }
            protected override void Layout()
            {
                base.Layout();

                Rectangle rec0 = GH_Convert.ToRectangle(Bounds);

                rec0.Height += 42;

                Rectangle rec1 = rec0;
                rec1.X = rec0.Left + 1;
                rec1.Y = rec0.Bottom - 42;
                rec1.Width = (rec0.Width) / 3 + 1;
                rec1.Height = 22;
                rec1.Inflate(-2, -2);

                Rectangle rec2 = rec1;
                rec2.X = rec1.Right + 2;

                Rectangle rec3 = rec2;
                rec3.X = rec2.Right + 2;

                Rectangle rec4 = rec1;
                rec4.Y = rec1.Bottom + 2;
                rec4.Width = rec0.Width - 6;

                Bounds = rec0;
                BoundsAllButtons = rec0;
                ButtonBounds = rec1;
                ButtonBounds2 = rec2;
                ButtonBounds3 = rec3;
                ButtonBounds4 = rec4;

            }

            GH_Palette xColor = GH_Palette.Black;
            GH_Palette yColor = GH_Palette.Black;
            GH_Palette zColor = GH_Palette.Black;
            GH_Palette rxColor = GH_Palette.Black;

            private Rectangle BoundsAllButtons { get; set; }
            private Rectangle ButtonBounds { get; set; }
            private Rectangle ButtonBounds2 { get; set; }
```

```
        private Rectangle ButtonBounds3 { get; set; }
        private Rectangle ButtonBounds4 { get; set; }

        protected override void Render(GH_Canvas canvas, Graphics
            graphics, GH_CanvasChannel channel)
        {
            base.Render(canvas, graphics, channel);
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button =
                    GH_Capsule.CreateTextCapsule(ButtonBounds,
                    ButtonBounds, xColor, "X", 3, 0);
                button.Render(graphics, Selected, false, false);
                button.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button2 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds2,
                    ButtonBounds2, yColor, "Y", 2, 0);
                button2.Render(graphics, Selected, Owner.Locked,
                    false);
                button2.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button3 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds3,
                    ButtonBounds3, zColor, "Z", 2, 0);
                button3.Render(graphics, Selected, Owner.Locked,
                    false);
                button3.Dispose();
            }
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button4 =
                    GH_Capsule.CreateTextCapsule(ButtonBounds4,
                    ButtonBounds4, rxColor, "Fix Rotation", 2, 0);
                button4.Render(graphics, Selected, Owner.Locked,
                    false);
                button4.Dispose();
            }
        }

```

```
            public override GH_ObjectResponse
                RespondToMouseDown(GH_Canvas sender, GH_CanvasMouseEvent
                e)
            {
                if (e.Button == MouseButtons.Left)
                {
                    RectangleF rec = ButtonBounds;
                    if (rec.Contains(e.CanvasLocation))
                    {
                        switchColor("X");
                    }
                    rec = ButtonBounds2;
                    if (rec.Contains(e.CanvasLocation))
                    {
                        switchColor("Y");
                    }
                    rec = ButtonBounds3;
                    if (rec.Contains(e.CanvasLocation))
                    {
                        switchColor("Z");
                    }
                    rec = ButtonBounds4;
                    if (rec.Contains(e.CanvasLocation))
                    {
                        switchColor("RX");
                    }
                    rec = BoundsAllButtons;
                    if (rec.Contains(e.CanvasLocation))
                    {
                        if (xColor == GH_Palette.Black) {
                            BDCComponent.setBDC("X", 0); }
                        if (xColor == GH_Palette.Grey) {
                            BDCComponent.setBDC("X", 1); }
                        if (yColor == GH_Palette.Black) {
                            BDCComponent.setBDC("Y", 0); }
                        if (yColor == GH_Palette.Grey) {
                            BDCComponent.setBDC("Y", 1); }
                        if (zColor == GH_Palette.Black) {
                            BDCComponent.setBDC("Z", 0); }
                        if (zColor == GH_Palette.Grey) {
                            BDCComponent.setBDC("Z", 1); }
                        if (rxColor == GH_Palette.Black) {
                            BDCComponent.setBDC("RX", 0); }
                        if (rxColor == GH_Palette.Grey) {
```

```
                                BDCComponent.setBDC("RX", 1); }
                            sender.Refresh();
                            Owner.ExpireSolution(true);
                        }
                        return GH_ObjectResponse.Handled;
                    }
                    return base.RespondToMouseDown(sender, e);
                }

                private void switchColor(string button)
                {
                    if (button == "X")
                    {
                        if (xColor == GH_Palette.Black) { xColor =
                            GH_Palette.Grey; }
                        else { xColor = GH_Palette.Black; }
                    }
                    else if (button == "Y")
                    {
                        if (yColor == GH_Palette.Black) { yColor =
                            GH_Palette.Grey; }
                        else { yColor = GH_Palette.Black; }
                    }
                    else if (button == "Z")
                    {
                        if (zColor == GH_Palette.Black) { zColor =
                            GH_Palette.Grey; }
                        else { zColor = GH_Palette.Black; }
                    }
                    else if (button == "RX")
                    {
                        if (rxColor == GH_Palette.Black) { rxColor =
                            GH_Palette.Grey; }
                        else { rxColor = GH_Palette.Black; }
                    }
                }
            }
        }
}
```

Deformed Shell component

```
using System;
using System.Collections.Generic;
using Grasshopper.Kernel;
using Rhino.Geometry;
using System.Drawing;
using Grasshopper.GUI.Canvas;
using System.Windows.Forms;
using Grasshopper.GUI;
using MathNet.Numerics.LinearAlgebra;

namespace Shell
{
    public class DeformedGeometry : GH_Component
    {
        public DeformedGeometry()
          : base("DeformedShell", "DefS",
              "Displays the deformed shell, with or without coloring",
              "Koala", "Shell")
        {
        }

        //Initialize startcondition and polynomial order
        static bool startDef = true;
        static bool setColor = false;
        static bool X = false;
        static bool Y = false;
        static bool VonMisesButton = false;
        static bool RX = false;
        static bool RY = false;

        //Method to allow c hanging of variables via GUI (see Component
            Visual)
        public static void setToggles(string s, bool i)
        {
            if (s == "Run")
            {
                startDef = i;
            }
            if (s == "setColor")
            {
                setColor = i;
            }
```

```
            if (s == "X")
            {
                X = i;
            }
            if (s == "Y")
            {
                Y = i;
            }
            if (s == "VonMises")
            {
                VonMisesButton = i;
            }
            if (s == "RX")
            {
                RX= i;
            }
            if (s == "RY")
            {
                RY = i;
            }
        }

        public override void CreateAttributes()
        {
            m_attributes = new Attributes_Custom(this);
        }

        protected override void
            RegisterInputParams(GH_Component.GH_InputParamManager
            pManager)
        {
            pManager.AddNumberParameter("Deformation", "Def",
                "Deformations from ShellCalc", GH_ParamAccess.list);
            pManager.AddNumberParameter("Stresses", "Stress", "Stresses
                from ShellCalc", GH_ParamAccess.list, new List<double> {
                0 });
            pManager.AddMeshParameter("Mesh", "M", "Input Geometry (Mesh
                format)", GH_ParamAccess.item);
            pManager.AddNumberParameter("Scale", "S", "The Scale Factor
                for Deformation", GH_ParamAccess.item, 10);
            pManager.AddNumberParameter("Yield Strength", "YieldS", "The
                Yield Strength in MPa", GH_ParamAccess.list, new
                List<double> { 0, 0 });
        }
```

```
        protected override void RegisterOutputParams(GH_Component.GH_OutputParamManager pManager)
        {
            pManager.AddMeshParameter("Deformed Geometry", "Def.G.", "Deformed Geometry as mesh", GH_ParamAccess.item);
            pManager.AddNumberParameter("Von Mises stress", "VMS", "The Von Mises yield criterion", GH_ParamAccess.list);
        }

        protected override void SolveInstance(IGH_DataAccess DA)
        {
            #region Fetch input
            //Expected inputs and outputs
            List<double> def = new List<double>();
            List<double> stresses = new List<double>();
            List<double> VonMises = new List<double>();
            Mesh mesh = new Mesh();
            double scale = 10;
            List<double> yieldStrength = new List<double>();
            List<Line> defGeometry = new List<Line>();
            List<Point3d> defPoints = new List<Point3d>();

            int[] h = new int[] { 0, 0, 0 };
            //Set expected inputs from Indata
            if (!DA.GetDataList(0, def)) return;
            if (!DA.GetDataList(1, stresses)) return;
            if (!DA.GetData(2, ref mesh)) return;
            if (!DA.GetData(3, ref scale)) return;
            if (!DA.GetDataList(4, yieldStrength)) return;
                #endregion

            #region Decompose Mesh and initiate the new deformed mesh defmesh

            List<Point3d> vertices = new List<Point3d>();
            List<MeshFace> faces = new List<MeshFace>();

            foreach (var vertice in mesh.Vertices)
            {
                vertices.Add(vertice);
            }
            foreach (var face in mesh.Faces)
```

```
            {
                faces.Add(face);
            }

            Mesh defmesh = new Mesh();

            defmesh.Faces.AddFaces(mesh.Faces); // new mesh without vertices

            #endregion

            if (stresses.Count > 0 && !(stresses.Count == 1 && stresses[0] == 0))
            {
                #region Von Mises
                for (int j = 0; j < faces.Count; j++)
                {
                    double sigma11 = stresses[j * 6];
                    if (sigma11 >= 0)
                    {
                        sigma11 += Math.Abs(stresses[j * 6 + 3]);
                    }
                    else
                    {
                        sigma11 += -Math.Abs(stresses[j * 6 + 3]);
                    }

                    double sigma22 = stresses[j * 6 + 1];
                    if (sigma22 >= 0)
                    {
                        sigma22 += Math.Abs(stresses[j * 6 + 4]);
                    }
                    else
                    {
                        sigma22 += -Math.Abs(stresses[j * 6 + 4]);
                    }

                    double sigma12 = stresses[j * 6 + 2];
                    if (sigma12 >= 0)
                    {
                        sigma12 += Math.Abs(stresses[j * 6 + 5]);
                    }
                    else
                    {
```

```
                    sigma12 += -Math.Abs(stresses[j * 6 + 5]);
                }

                VonMises.Add(Math.Sqrt(sigma11 * sigma11 - sigma11 *
                    sigma22 + sigma22 * sigma22 + 3 * sigma12 *
                    sigma12));
            }

            #endregion
        }

        if (startDef)
        {
            #region apply deformations to vertices and add them to
                defmesh

            List<Point3d> new_vertices = new List<Point3d>(); // list
                of translated vertices
            int i = 0;

            foreach (var p in vertices)
            {
                new_vertices.Add(new Point3d(p.X + def[i]*scale, p.Y
                    + def[i + 1]*scale, p.Z + def[i + 2]*scale));
                i += 3;
            }

            defmesh.Vertices.AddVertices(new_vertices);
            #endregion

            int dimension = 123;
            if (X)
            {
                dimension = 0;
            }
            else if (Y)
            {
                dimension = 1;
            }
            else if (VonMisesButton)
            {
                dimension = 7;
            }
            else if (RX)
```

```
            {
                dimension = 3;
            }
            else if (RY)
            {
                dimension = 4;
            }

            Mesh coloredDefMesh = defmesh.DuplicateMesh();
            if (setColor && (stresses.Count > 1 || (stresses.Count ==
                1 && stresses[0] != 0) || VonMises.Count > 1 ||
                (VonMises.Count == 1 && VonMises[0] != 0)) &&
                (dimension < 8))
            {
                // Direction can be 0 -> x ...
                SetMeshColors(defmesh, stresses, VonMises,
                    new_vertices, faces, dimension, yieldStrength,
                    out coloredDefMesh);
            }

            //Set output data
            DA.SetData(0, coloredDefMesh);
            DA.SetDataList(1, VonMises);
        }
    }   //End of main program

    private void SetMeshColors(Mesh meshIn, List<double> stresses,
        List<double> VonMises, List<Point3d> vertices, List<MeshFace>
        faces, int direction, List<double> yieldStrength, out Mesh
        meshOut)
    {
        meshOut = meshIn.DuplicateMesh();

        List<int> R = new List<int>(faces.Count);
        List<int> G = new List<int>(faces.Count);
        List<int> B = new List<int>(faces.Count);
        int[,] facesConnectedToVertex = new int[faces.Count,3];

        double max = 0;
        double min = 0;

        if (yieldStrength.Count == 1 && yieldStrength[0] > 1)
        {
            max = yieldStrength[0];
```

```
                min = -yieldStrength[0];
            }
            else if ((yieldStrength.Count == 1 && yieldStrength[0] == 0) || (yieldStrength[0] == 0 && yieldStrength[1] == 0) || yieldStrength.Count == 0)
            {
                for (int i = 0; i < stresses.Count / 6; i++)
                    {
                        double stress;
                        if (direction < 6)
                        {
                            stress = stresses[i * 6 + direction];
                        }
                        else
                        {
                            stress = VonMises[i];
                        }
                        if (stress > max)
                        {
                            max = stress;
                    }
                    else if (stress < min)
                    {
                        min = stress;
                    }
                }
            }
            else
            {
                if (yieldStrength[0] >= 0 && yieldStrength[1] <= 0)
                {
                    max = yieldStrength[0];
                    min = yieldStrength[1];
                }
                else if (yieldStrength[1] >= 0 && yieldStrength[0] <= 0)
                {
                    max = yieldStrength[1];
                    min = yieldStrength[0];
                }
                else
                {
                    AddRuntimeMessage(GH_RuntimeMessageLevel.Warning, "Warning message here");
                }
```

```
}



List<double> colorList = new List<double>();

for (int i = 0; i < faces.Count; i++)
{
    double stress;
    if (direction < 6)
    {
        stress = stresses[i*6+direction];
    }
    else
    {
        stress = VonMises[i];
    }

    R.Add(0);
    G.Add(0);
    B.Add(0);

    if (stress >= max)
    {
        R[i] = 255;
    }
    else if (stress >= max*0.5 && max != 0)
    {
        R[i] = 255;
        G[i] = Convert.ToInt32(Math.Round(255 * (1 - (stress
             - max * 0.5) / (max * 0.5))));
    }
    else if (stress < max*0.5 && stress >= 0 && max != 0)
    {
        G[i] = 255;
        R[i] = Convert.ToInt32(Math.Round(255 * (stress) /
             (max * 0.5)));
    }
    else if (stress < 0 && stress > min*0.5 && min != 0)
    {
        G[i] = 255;
        B[i] = Convert.ToInt32(Math.Round(255 * (stress) /
             (min * 0.5)));
```

```
                }
                else if (stress <= min*0.5 && min != 0 && stress > min)
                {
                    B[i] = 255;
                    G[i] = Convert.ToInt32(Math.Round(255 * (1 - (stress
                        - min * 0.5) / (min * 0.5))));
                }
                else if (stress <= min)
                {
                    B[i] = 255;
                }
            }

            for (int i = 0; i < vertices.Count; i++)
            {
                List<int> vertex = new List<int>();
                int vR = 0, vG = 0, vB = 0;
                for (int j = 0; j < faces.Count; j++)
                {
                    if (faces[j].A == i || faces[j].B == i || faces[j].C
                        == i)
                    {
                        vertex.Add(j);
                    }
                }
                for (int j = 0; j < vertex.Count; j++)
                {
                    vR += R[vertex[j]];
                    vG += G[vertex[j]];
                    vB += B[vertex[j]];
                }
                vR /= vertex.Count;
                vG /= vertex.Count;
                vB /= vertex.Count;

                meshOut.VertexColors.Add(vR, vG, vB);
            }
        }

        private List<Point3d> CreatePointList(List<Line> geometry)
        {
            List<Point3d> points = new List<Point3d>();

            for (int i = 0; i < geometry.Count; i++) //adds every point
```

```
                unless it already exists in list
            {
                Line l1 = geometry[i];
                if (!points.Contains(l1.From))
                {
                    points.Add(l1.From);
                }
                if (!points.Contains(l1.To))
                {
                    points.Add(l1.To);
                }
            }

            return points;
        }

        protected override System.Drawing.Bitmap Icon
        {
            get
            {
                return Properties.Resources.Draw1;
            }
        }

        public override Guid ComponentGuid
        {
            get { return new
                Guid("4b28fb40-2e66-4d19-a629-c630c079725a"); }
        }


        /// Component Visual//
        public class Attributes_Custom :
            Grasshopper.Kernel.Attributes.GH_ComponentAttributes
        {
            public Attributes_Custom(GH_Component owner) : base(owner) { }
            protected override void Layout()
            {
                base.Layout();

                Rectangle rec0 = GH_Convert.ToRectangle(Bounds);

                if (setColor)
                {
```

```
        rec0.Height += 82;
    }
    else
    {
        rec0.Height += 22;
    }

    Rectangle rec1 = rec0;
    rec1.X = rec0.Left + 1;

    if (setColor)
    {
        rec1.Y = rec0.Bottom - 82;
    }
    else
    {
        rec1.Y = rec0.Bottom - 22;
    }
    rec1.Width = (rec0.Width) / 2;
    rec1.Height = 22;
    rec1.Inflate(-2, -2);

    Rectangle rec2 = rec1;
    rec2.X = rec1.Right + 2;

    Rectangle rec3 = rec2;
    rec3.X = rec1.X;
    rec3.Y = rec1.Bottom + 2;

    Rectangle rec4 = rec3;
    rec4.X = rec3.X;
    rec4.Y = rec3.Bottom + 2;

    Rectangle rec5 = rec3;
    rec5.X = rec4.X;
    rec5.Y = rec4.Bottom + 2;

    Rectangle rec6 = rec3;
    rec6.X = rec5.Right + 2;
    rec6.Y = rec3.Bottom + 2;

    Rectangle rec7 = rec3;
    rec7.X = rec6.X;
    rec7.Y = rec6.Bottom + 2;
```

```

            Bounds = rec0;
            ButtonBounds = rec1;
            ButtonBounds1 = rec2;
            ButtonBounds2 = rec3;
            ButtonBounds3 = rec4;
            ButtonBounds4 = rec5;
            ButtonBounds5 = rec6;
            ButtonBounds6 = rec7;

        }

        GH_Palette displayed = GH_Palette.Black;
        GH_Palette setcolor = GH_Palette.Grey;
        GH_Palette xColor = GH_Palette.Grey;
        GH_Palette yColor = GH_Palette.Grey;
        GH_Palette VonMisesColor = GH_Palette.Grey;
        GH_Palette rxColor = GH_Palette.Grey;
        GH_Palette ryColor = GH_Palette.Grey;

        private Rectangle ButtonBounds { get; set; }
        private Rectangle ButtonBounds1 { get; set; }
        private Rectangle ButtonBounds2 { get; set; }
        private Rectangle ButtonBounds3 { get; set; }
        private Rectangle ButtonBounds4 { get; set; }
        private Rectangle ButtonBounds5 { get; set; }
        private Rectangle ButtonBounds6 { get; set; }

        protected override void Render(GH_Canvas canvas, Graphics
            graphics, GH_CanvasChannel channel)
        {
            base.Render(canvas, graphics, channel);
            if (channel == GH_CanvasChannel.Objects)
            {
                GH_Capsule button;
                if (startDef == false)
                {
                    button =
                        GH_Capsule.CreateTextCapsule(ButtonBounds,
                        ButtonBounds, displayed, "Hidden", 3, 0);
                    button.Render(graphics, Selected, Owner.Locked,
                        false);
                    button.Dispose();
                }
```

```
                else
                {
                    button =
                        GH_Capsule.CreateTextCapsule(ButtonBounds,
                        ButtonBounds, displayed, "Displayed", 3, 0);
                    button.Render(graphics, Selected, Owner.Locked,
                        false);
                    button.Dispose();
                }
                if (setColor == true)
                {
                    GH_Capsule button2 =
                        GH_Capsule.CreateTextCapsule(ButtonBounds1,
                        ButtonBounds1, setcolor, "Colored", 2, 0);
                    button2.Render(graphics, Selected, Owner.Locked,
                        false);
                    button2.Dispose();
                }
                else
                {
                    GH_Capsule button2 =
                        GH_Capsule.CreateTextCapsule(ButtonBounds1,
                        ButtonBounds1, setcolor, "Uncolored", 2, 0);
                    button2.Render(graphics, Selected, Owner.Locked,
                        false);
                    button2.Dispose();
                }
                if (setColor == true)
                {
                    GH_Capsule button3 =
                        GH_Capsule.CreateTextCapsule(ButtonBounds2,
                        ButtonBounds2, xColor, "X Stresses", 2, 0);
                    button3.Render(graphics, Selected, Owner.Locked,
                        false);
                    button3.Dispose();
                }
                if (setColor == true)
                {
                    GH_Capsule button4 =
                        GH_Capsule.CreateTextCapsule(ButtonBounds3,
                        ButtonBounds3, yColor, "Y Stresses", 2, 0);
                    button4.Render(graphics, Selected, Owner.Locked,
                        false);
                    button4.Dispose();
```

```
                }
                if (setColor == true)
                {
                    GH_Capsule button5 =
                        GH_Capsule.CreateTextCapsule(ButtonBounds4,
                        ButtonBounds4, VonMisesColor, "Von Mises", 2,
                        0);
                    button5.Render(graphics, Selected, Owner.Locked,
                        false);
                    button5.Dispose();
                }
                if (setColor == true)
                {
                    GH_Capsule button6 =
                        GH_Capsule.CreateTextCapsule(ButtonBounds5,
                        ButtonBounds5, rxColor, "RX Stresses", 2, 0);
                    button6.Render(graphics, Selected, Owner.Locked,
                        false);
                    button6.Dispose();
                }
                if (setColor == true)
                {
                    GH_Capsule button7 =
                        GH_Capsule.CreateTextCapsule(ButtonBounds6,
                        ButtonBounds6, ryColor, "RY Stresses", 2, 0);
                    button7.Render(graphics, Selected, Owner.Locked,
                        false);
                    button7.Dispose();
                }
            }
        }

        public override GH_ObjectResponse
            RespondToMouseDown(GH_Canvas sender, GH_CanvasMouseEvent
            e)
        {
            if (e.Button == MouseButtons.Left)
            {
                RectangleF rec = ButtonBounds;
                if (rec.Contains(e.CanvasLocation))
                {
                    switchColor("Run");
                }
                rec = ButtonBounds1;
```

```
            if (rec.Contains(e.CanvasLocation))
            {
                switchColor("setColor");
            }
            rec = ButtonBounds2;
            if (rec.Contains(e.CanvasLocation))
            {
                switchColor("X");
            }
            rec = ButtonBounds3;
            if (rec.Contains(e.CanvasLocation))
            {
                switchColor("Y");
            }
            rec = ButtonBounds4;
            if (rec.Contains(e.CanvasLocation))
            {
                switchColor("VonMises");
            }
            rec = ButtonBounds5;
            if (rec.Contains(e.CanvasLocation))
            {
                switchColor("RX");
            }
            rec = ButtonBounds6;
            if (rec.Contains(e.CanvasLocation))
            {
                switchColor("RY");
            }

            if (displayed == GH_Palette.Black) { DeformedGeometry.setToggles("Run", true); }
            if (displayed == GH_Palette.Grey) { DeformedGeometry.setToggles("Run", false); }
            if (setcolor == GH_Palette.Black) { DeformedGeometry.setToggles("setColor", true); }
            if (setcolor == GH_Palette.Grey) { DeformedGeometry.setToggles("setColor", false); }
            if (xColor == GH_Palette.Black) { DeformedGeometry.setToggles("X", true); }
            if (xColor == GH_Palette.Grey) { DeformedGeometry.setToggles("X", false); }
            if (yColor == GH_Palette.Black) { DeformedGeometry.setToggles("Y", true); }
```

```
                if (yColor == GH_Palette.Grey) { DeformedGeometry.setToggles("Y", false); }
                if (VonMisesColor == GH_Palette.Black) { DeformedGeometry.setToggles("VonMises", true); }
                if (VonMisesColor == GH_Palette.Grey) { DeformedGeometry.setToggles("VonMises", false); }
                if (rxColor == GH_Palette.Black) { DeformedGeometry.setToggles("RX", true); }
                if (rxColor == GH_Palette.Grey) { DeformedGeometry.setToggles("RX", false); }
                if (ryColor == GH_Palette.Black) { DeformedGeometry.setToggles("RY", true); }
                if (ryColor == GH_Palette.Grey) { DeformedGeometry.setToggles("RY", false); }
                sender.Refresh();
                Owner.ExpireSolution(true);
                return GH_ObjectResponse.Handled;

            }
            return base.RespondToMouseDown(sender, e);
        }

        private void switchColor(string button)
        {
            if (button == "Run")
            {
                if (displayed == GH_Palette.Black) { displayed = GH_Palette.Grey; }
                else { displayed = GH_Palette.Black; }
            }
            if (button == "setColor")
            {
                if (setcolor == GH_Palette.Black)
                {
                    setcolor = GH_Palette.Grey;
                    xColor = GH_Palette.Grey;
                    yColor = GH_Palette.Grey;
                    VonMisesColor = GH_Palette.Grey;
                    rxColor = GH_Palette.Grey;
                    ryColor = GH_Palette.Grey;
                }
                else { setcolor = GH_Palette.Black; }
            }
            if (button == "X" && setcolor == GH_Palette.Black)
```

```
                {
                    if (xColor == GH_Palette.Black) { xColor =
                        GH_Palette.Grey; }
                    else
                    {
                        xColor = GH_Palette.Black;
                        yColor = GH_Palette.Grey;
                        VonMisesColor = GH_Palette.Grey;
                        rxColor = GH_Palette.Grey;
                        ryColor = GH_Palette.Grey;
                    }
                }
                if (button == "Y" && setcolor == GH_Palette.Black)
                {
                    if (yColor == GH_Palette.Black) { yColor =
                        GH_Palette.Grey; }
                    else
                    {
                        yColor = GH_Palette.Black;
                        xColor = GH_Palette.Grey;
                        VonMisesColor = GH_Palette.Grey;
                        rxColor = GH_Palette.Grey;
                        ryColor = GH_Palette.Grey;
                    }
                }
                if (button == "VonMises" && setcolor == GH_Palette.Black)
                {
                    if (VonMisesColor == GH_Palette.Black) {
                        VonMisesColor = GH_Palette.Grey; }
                    else
                    {
                        VonMisesColor = GH_Palette.Black;
                        xColor = GH_Palette.Grey;
                        yColor = GH_Palette.Grey;
                        rxColor = GH_Palette.Grey;
                        ryColor = GH_Palette.Grey;
                    }
                }
                if (button == "RX" && setcolor == GH_Palette.Black)
                {
                    if (rxColor == GH_Palette.Black) { rxColor =
                        GH_Palette.Grey; }
                    else
                    {
```

```
                        xColor = GH_Palette.Grey;
                        yColor = GH_Palette.Grey;
                        VonMisesColor = GH_Palette.Grey;
                        rxColor = GH_Palette.Black;
                        ryColor = GH_Palette.Grey;
                    }
                }
                if (button == "RY" && setcolor == GH_Palette.Black)
                {
                    if (ryColor == GH_Palette.Black) { ryColor =
                        GH_Palette.Grey; }
                    else
                    {
                        xColor = GH_Palette.Grey;
                        yColor = GH_Palette.Grey;
                        VonMisesColor = GH_Palette.Grey;
                        rxColor = GH_Palette.Grey;
                        ryColor = GH_Palette.Black;
                    }
                }
            }
        }
    }
}
```