NTNU
Norwegian University of
Science and Technology

The Parametric design of adaptive joints

Helge Roksvåg Mæhle

Master of Science in Civil and Environmental Engineering
Submission date: June 2017
Supervisor: Anders Rönnquist, KT

Norwegian University of Science and Technology
Department of Structural Engineering

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

ACCESSIBILITY

OPEN

MASTER THESIS 2017

SUBJECT AREA: Structural Engineering	DATE: 11.06.2017	NO. OF PAGES:

TITLE:

The Parametric design of adaptive joints

BY:

Helge Roksvåg Mæhle

SUMMARY:
Building information modeling (BIM) has had a big impact on modern engineering. Through the information contained in the models, engineers are able to interpret and organize a vast amount of data. During a conceptual design phase, a digital platform based on a Knowledge-based engineering (KBE) model helps in eliminating poor designs, enhancing the process. This paper examines the possibility of direct contact between a computer aided three-dimensional interactive application (e.g. CATIA) and a Finite Element Software. The goal is to create a method for quick export of building information model from common CAD-software to an advanced finite element analysis. An approach is derived in which the necessary data from the pre-processor phase is generated for a finite element simulation.

RESPONSIBLE TEACHER: Professor Anders Rönnquist

SUPERVISOR(S): Marcin Luczkowski

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

Summary

Building information modeling (BIM) has had a big impact on modern engineering. Through the information contained in the models, engineers are able to interpret and organize a vast amount of data. During a conceptual design phase, a digital platform based on a Knowledge-based engineering (KBE) model helps in eliminating poor designs, enhancing the process. This paper examines the possibility of direct contact between a computer aided three-dimensional interactive application (e.g. CATIA) and a Finite Element Software. The goal is to create a method for quick export of building information model from common CAD-software to an advanced finite element analysis. An approach is derived in which the necessary data from the pre-processor phase is generated for a finite element simulation.

Sammenrag

Bygnings informasjons modellering (BIM) har hatt stor påvirkning på moderne prosjektering. Gjennom informasjonen i modellene er ingeniørene i stand til å tyde og organisere store mengder data. Under konseptuell design så vil en veletablert digital plattform basert på kunnskapsbasert prosjektering bidra til å eliminere svake løsninger og fremme prosessen. Denne oppgaven undersøker muligheten i å oppnå direkte kontakt mellom en dataassistert tredimensjonal interaktiv applikasjon (e.g. CATIA) og en endelig element programvare. Målet er å skape en metode for kjapp overføring av bygnings informasjons modell fra CAD-programvare til en avansert endelig element analyse. En fremgangsmåte ble utledet, som generere den nødvendige dataen fra en pre-prosesseringen fase for en endelig element simulering.

Problem Description

Engineering Architecture - The Parametric design of adaptive joints

Create a parametric model of a steel structures which includes custom modules that gives a real-time capacity verification of welded joints according to EC3 and generate data for a FEA of the cruical connection. Also, investigate how the boundary conditions given by the welded connection affects the form finding process and how the form finding affects forces in the connections.

Key words:
-Analysis welded steel connection
-Eurocode 3
-Python
-Parametric modeling
-Knowledge based design
-Rhino with Grasshopper
-ABAQUS
-Computation time

Preface

This dissertation, written in the spring of 2017, is the Master thesis which concludes my Master of Science degree at the Norwegian University of Science and Technology, department of structural engineering.

Throughout the length of my studies I've registered that the industry have moved more towards digital solutions, building information modeling is close to becoming the norm for planing and managing projects. Which is one of the reasons why I chose this kind of problem for my master thesis, It's most likely that I will be working with digital models at on point in my career.

The method Parametric design used for this problem is a powerful application which makes one capable of deriving ones own modules and their interrelation. I've worked actively with building information modeling prior to this thesis, and have observed what can be done with the assistance of parametric modeling. Complex project have been made possible through a collection of well defined lines of code, which is impressive to say the least. Mastering this to a certain degree is one of my goal for this problem, since it will give access to new engineering possibilities.

I wish to thank Professor Anders Rönnquist for our discussions during this spring, and to Marcin Luczkowski for making time to introduce me to the software and for his guidance throughout this process. I am truly grateful for all the help I've got this spring.

Contents

List of Figures

Abbreviations

BIM	=	Building Information Model
CAD	=	Computer Aided Design
CATIA	=	Computer Aided Three-Dimensional Interactive Application
CNC	=	Computer Numerical Control
DOF	=	Degrees of Freedom
EC	=	Eurocode
FEA	=	Finite Element Analysis
FEM	=	Finite Element Method
GH	=	Grasshopper
GUI	=	Graphical User Interface
INP-file	=	Input file
OOP	=	Object-Oriented Programming
KBE	=	Knowledge-Based Engineering
SDK	=	Software Developers Kit

1 Introduction

This dissertation examines the possibility of direct connection between a computer aided three dimensional interactive application (CATIA) and a Finite Element Software. The goal is to create a method for quick exporting building information model (BIM) from common CAD-software to a advanced finite element analysis. An approach which generates the necessary data from the pre-processor phase of the finite element program is derived using Knowledge-based engineering (KBE) model.

1.1 Parametric Modeling

A parametric model is a geometrical structure/model which is build up using parameters, a quantifiable value which affects the characteristics of a system. By coupling parameters using mathematical and logical definitions, it's possible to record processes which recreates given actions. Through adjusting the inputs of a definition, one changes the characteristics of the action. In CAD, parametric modeling is to register a designs features in order to capture its intent, making it possible to recreate its behavior. This is referred to as creating classes of geometry. The example can be modeling of the IPE-beams. In traditional CAD geometrical properties are set for each dimension manually, while in a parametric model these values can be associated to a single parameter, e.g. the section type, which changes all the parameters linked to it automatically. This dependency between parameters leaves the model with a range of options which are easily accessed and changeable.

Parametric modeling is not exclusively used for physical models, it's also common for organizational purposes such as cost and time estimations. In building information models, parameters are used actively in order to store and manage large amount of data efficiently. Objects in the model have parameters which contains information about its discipline, properties, cost, time to install and etc. This information can be quantified and categorized, thus allowing the use of filtering methods. By filtering the data, engineers are able to extract the information which meet certain criteria, information which can be interpreted directly or be used as input in calculations or simulations. The digital approach is more efficient and accurate than going through this information manually.

1.2 Why Parametric Modeling

Parametric models are able to capture the characteristics of design through its definition. The characteristics changes according to the input values. By systematical going through possible inputs, manually or with the assistance of a code, it's possible to discover an optimal solution for a design problem. Typical kinds of operations are optimization methods and generating data. Optimization algorithms maps the behavior of a model by assigning the parameters a value, then it analyzes the new models characteristics before it reassigns a value according to a given objective. Today, optimizations are commonly used in order to maximize the utilization of structural elements minimizing the cross-sectional area[5]. Given certain constraints, an algorithm can generate data which follows a distinct pattern to create the basis for CNC fabrication[12]. For designs which contains vast number of components it becomes to cumbersome to create the necessary basis for fabrication manually. The use of algorithms is the most efficient option there is. The terms design to production and digital fabrication is often associated with this kind of automatized processes.

Using parametric to capture design are commonly used by manufacturer to distribute information about their products. Through the definition they're able to create models which display the features of the product they can produce and prohibit to display those they can't. These models can then be integrated in BIM and from there it provides users with real time information about the products specifications. This is an efficient way for the manufacturer to inform its customers of what they can and cannot deliver.

Many CAD software today offers designers the opportunity to create and implement their own procedures and commands, allowing them to create their own components and define their own inter-component behavior[2].

Parametric modeling is not without its cons. The associated constraints can prohibits designers to investigate possible solutions during the conceptional design phase making it redundant. At this phase manual modeling might be more suitable since it doesn't restrict the designer to a limited range of possible designs. And also, making large designs integrated into one parametric model results in a costly calculation time. A single adjustment of a parameter results that the whole model is recalculated, this is overcome by dividing the design into multiple sub routines or a hybrid between direct and parametric modeling.

1.3 Parametric Design past projects

In order to show how parametric modeling is used in practice, this section gives a short summary of some few projects where parametric played a crucial role.

The "Sequential roof" project for the new "Arch_Tec_Lab" building of the Institute of Technology in Architecture is a roof structure consisting of 168 individual timber trusses, all of which where designed individually in order to create the desired shape of the structure (fig.1). The elements of the trusses is built up by softwood timber with linear geometry and notch-free joints, which where assembled using a fully automated CNC woodworking technique. In order to design, analyze and detail the roof, the design group Gramazio Kohle Research at ETH Zurioch created an scheme which was a four step approach: First generate a model, then run a structural analyzes on it, followed by defining nail pattern at the joints and finally evaluate the results and do necessary modifications. In addition to being governed by the geometry of the roof, the algorithm had to account for building regulations and fabrication restriction in the design. Due to the sheer amount of elements of the structure (48,634 timber slats), the calculation time was costly using approximately 24 hours of work for each iteration. So the algorithm was developed and modified in to reduce amount of necessary iterations. This project shows the strength of fabrication-driven design, which could not be done as efficient without the use of an algorithm. More details about the project can be found in the article from the book Advances in Architectural Geometry 2016[3].

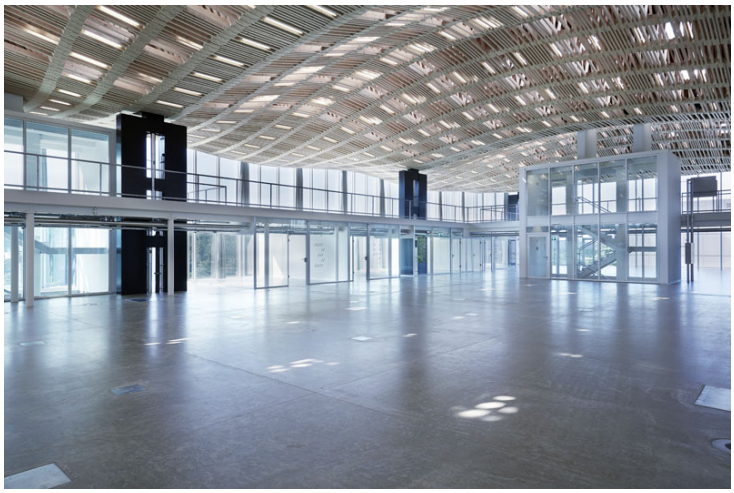


Figure 1: Interior side of the Sequential roof.

The Maggie's Centre in Manchester, a place of refuge for people struggling with cancer, is a single story building with a high rise roof structure which create a mezzanine level (fig.2). Due to the buildings purpose, timber was chosen as the main material because of its warmth. The structure of the roof consists of 17 identical frames in a repeated pattern over a 3m grid. A conventional approach of this kind of structural design would be to use glulam beams. Due to the structure being prone to loads, a parametric model was derived to create a lighter unique section and thus reduce self-weight load. This digital model combined with advanced manufacturing technology allowed the design team to find a find a design which carried the structure while reducing its the self-weight. More details about can be found in the article[9].

Figure 2: The roof structure of the Maggie's Centre in Manchester.

The Arup Group uses BIM actively in order to optimize the workflow in their projects. Information is the core of BIM and with the correct procedures an engineer is able to interpret and manage data more efficiently than before. Allowing workflows to be established when dealing with high amount of data or a new kind of data. Arup have developed an approach for transferring data directly between software not compatible which don't have BIM integrated. Through data script they translate the information from the on software to another in order to utilize both software in the design process. The aim is to optimize the work progress when working on complex programmes or irregular geometry. More details about can be found in the paper[7].

2 Software and Methods

In detail design it's sometimes necessary to do a FEA in order to control its capacities. Reason for analyzes might be due to complex geometry or it's required documentation. This means that the detail has to be modeled in a Finite Element software which is often time consuming. The approach derived in this paper exports a numerical model of geometry created in CAD to the solver in Abaqus by the means of an Input-file. It was developed using the modeling software Rhinoceros in combination with its plugin Grasshopper; The Grasshopper interface gives the user the opportunity to create parametric models within Rhinoceros. As input for the numerical model, results from Karamba (a Finite Element add-on for Grasshopper) was integrated into the analyzes. The approach is limited to static elastic linear problems using eight-node hexahedron, knows as eight-node brick elements. Steel was chosen as material due to its isotropic properties which enables the use of von mises to measure the stresses.

2.1 Softwares and Plugins

Abaqus is a Finite Element Analysis software used in multiple industries due to its modeling capabilities and the possibility to or to do customizations. The preprocessing in Abaqus are usually performed in Abqus/CAE, at this part of the software is where the numerical model is defined. The analyzes runs in Abaqus/Standard or Abaqus/Explicit depending on the type of problem; The solver Abaqus/Standard in is ideal for static and low-speed dynamic events, while Abaqus/Explicit simulates brief transient dynamic events. It is Abaqus/Standard which is employed in this paper.

Rhinoceros, often abbreviated as Rhino, is a CAD software frequently used by designers and architects. The Rhinois common used by designers thanks to NURBS, Non-Uniform Rational B-Splines, which are mathematically representations that can accurately create shapes as curves and freeform surfaces[10]. The flexibility and accuracy of NURBS allows the user to create precise representations of any curves and freeform surfaces, which is usable processes from illustration to manufacturing.

Grasshopper is a graphical algorithm editor for Rhino which provide users with a graphical interface to create programs know as definitions. All the information is laid out on the canvas in form of parameters and components and the data flows from the left to the right. Parameters is the data in the definition; These can either be containers which store existing information or

input parameters which are organized on the canvas. Components perform an action based on the input it receives. Grasshopper offers a variety of components which performs different tasks within different disciplines. The user defines the relation between parameters and components by connecting their nodes together using wires. Nodes is the access point for input or output information for each component on the canvas, while wires represent data passing through. The anatomy of a component is that the nodes on the left are input data while those on the right are output. This data on the canvas results in an algorithm which is called a definition. More details about the anatomy of Grasshopper definitions can be found in The Grasshopper Primer[10].

Karamba is a plugin for Grasshopper which runs real time FEA of 3D-Beams and shell on the Grasshopper canvas. The real time display of results gives the user an insight in the reactions and how it's affected by certain inputs. Karamba uses a limited amount of elements, approximately 10 000, it's mainly used for conceptional design in order to investigate possible design.

More information about the softwares and plugins can be found on their respective homepage.

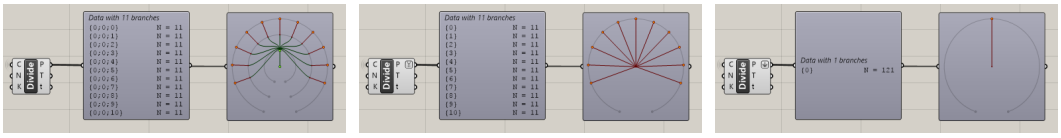
2.2 Scripting in Grasshopper

Grasshopper offers designers the possibility to create custom codes that runs through components on the canvas. The VB, Python and C# components gives the framework of their respective language while giving access to the SDK (Software Developers Kit) which includes libraries to call Rhino commands through syntaxes. With codes it's possible to create new procedures and functions not included in the Grasshopper package. In this paper the Python and C# components where used.

When creating components the user have to call which type of data that's being used, this is due to the Object-Oriented programming (OOP) paradigm of the languages. This means that for all inputs, variables and output the type has to be called at the time its created. This lets the component know which kind of data it's working on and it's able to use them in commands and find the corresponding syntaxes from the library. The framework of the Python script is not as strict on the calling of data as C# which means its better suited when drafting the code.

Grasshopper allows the use of Lists and Data Trees to store and transport

data over the canvas. These are arrays of specific data types such as Strings, Doubles and various geometric types which are called by its position in the array. The Data Tree differs from Lists since it's a hierarchical system of nested Lists called Branches and are assigned a path which tell Grasshopper where to find it. A common analogy of Data Tree Branches is the folder system of windows explorer, Data in the Tree is like the files and the Branches are the folders. The Data Tree takes a forms the paths according to how the script assembles it. To access the data of a Branch within multple sub-branches then each sub-index has to be called which can become cumbersome. Grasshopper offers some command to manipulate the Trees anatomy, two which are shown in fig.3. Simplify removes the sub-indices of the Branches leaving just a single index for the list to be called, while Flatten takes all the data in the Tree and store it in a single Branch.



(a) A Data Tree of Points. (b) Simplified Tree. (c) Flatten Tree.

Figure 3: Anatomy of Trees in Grasshopper.

2.3 The Input File

In FEA, the preprocessor phase results in a numerical model of the problem at hand which acts as the basis for the simulation. The INP file, or input file, is a text file which contains the numerical model derived from the preprocessor and it's the communication platform between the pre-processor and the simulator. It uses computer aided three-dimensional interactive applications open platform commands in order to create numerical models. Figure4 shows step by step a typical analyzes in Abaqus. Since the INP is a text file it is easily accessed through a text editor. In the editor one gets direct control of the model allowing manipulation without the use of the GUI in Abaqus/CAE. This is faster since the GUI follows strict procedures that's time consuming, but since there is no GUI it will be difficult to do adapt the data for bigger files. In this paper the whole INP file is created in Grasshopper giving an alternative GUI to work with.

The INP file consists of mainly three parts: The physical model, the solving method and the output request. The physical model contains the discretization information of the geometry which includes its nodes, element, material

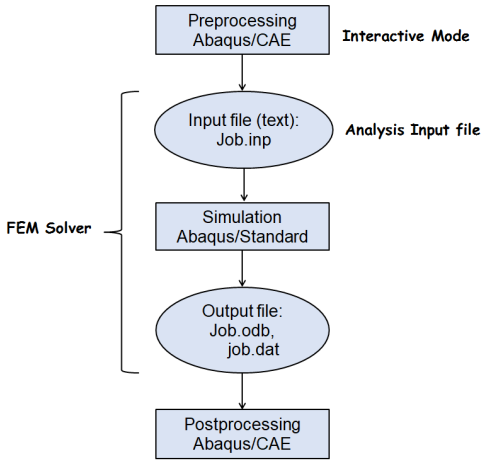


Figure 4: Basic Abaqus.

and how it's assembled. Solving method informs Abaqus of which kind of simulation its dealing with and the how the loads are applied. The output request gives the setting for how the information from the solver will be returned. This paper operates with a fixed setting for the output and will therefore not go into detail on how it's defined.

As seen in figure 5 the file consists of multiple distinct sections which are separated with double asterixes, examples from the figure are PARTS and ASSEMBLY. In each section it's possible to initiate an option by calling it with an asterix followed by a keyword. For example, to specify nodal coordinates it's *NODE and to specify element connectivity it's *ELEMENT. Note that when calling the element specification option it's followed by the sub-option *TYPE*, this is to let the solver know which type of element it's dealing with and how the nodes are connected. Following this format one is able to generate INP files for a given model, the challenge is to create a procedure which construct valid information to be used in the Abaqus solver.

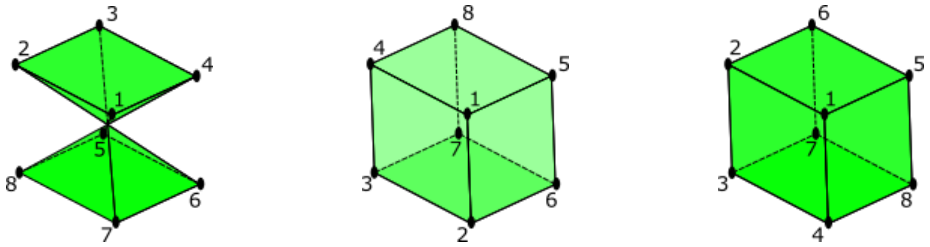
From the sub-option *TYPE*, Abaqus gets the command on how to connect the nodes the elements in the list. Element connectedness establishes the elements topology[1], for the eight-node hexahedron[6] the connectedness orientates the element faces and thus gives the face normals. These normals is used to determine the Jacobian[1][4] of the element. The Jacobian is a measure of the faces normals and their relation to each other. The Abaqus solver won't run if the Jacobian close to or below zero. Figure 6 shows different configurations of the eight-node hexahedron. In 6a the orientation of the

```
*Heading
<ID>
** Job name:<Job> Model name: <Model>
*Preprint, echo=NO, model=NO, history=NO, contact=NO
**
**PARTS
**
*Part, name= <Part_ID>
*Node

<<Node number,  X.Coord,  Z.Coord,  Y.Coord>>

*Element, type= <Type of Element>

<<Element number, Node nr.1, Node nr2, ..., Node nr8 >>

*Nset, nset=Set, generate
  1,   <Numer of Nodes>,          1
*Elset, elset=Set, generate
  1,   <Number of elements>,      1
**Section: End_section
*Solid Section, elset=Set, material= <Material>
,
*End Part
**
**
**ASSEMBLY
**
*Assembly, name = Assembly
**
*Instance, name = <Part_ID>, part= <ID>
*End Instance
**
<<Lists of part components>>
*End Assembly
```

```
**
**MATERIALS
**

<Material Properties>

**
** BOUNDARY CONDITIONS
**
** Name = <Support ID>, Type: Displacement/Rotation
*Boundary

<Conditions>

**------------------------------------------------------------------------
**
** STEP: <Applied Load>
**
*Step, name = <Load ID>, nlgeom=NO

<Load data>

**
** OUTPUT REQUEST
**
*Restart, write, frequency=0
**
** FIELD OUTPUT: F-Output-1
**
*Output, field, variable=PRESELECT
**
** HISTORY OUTPUT: H-Output-1
**
*Output, history, variable=PRESELECT
*End Step
```

Figure 5: The Structure of an INP file.

bottom face is oriented 180degrees away from the correct position (node 7 and node 5, and node 8 and node 6 should interchange) resulting in a badly shaped element. The face normals in 6b are oriented towards the elements center, resulting in a negative volume. The element in 6b has the correct topological mapping.



(a) Misshaped (b) Negative Volume (c) Correct

Figure 6: Sketch of multiple discretization of hexahedron.

The assembly section of the INP file contains information about the parts used in the model (instances) and how their components (nodes and elements) are restrained and/or loaded. In order to apply such conditions, sets containing the identification of entities is created and labeled to the given operation. The sets contains information of either node or elements, and the keyword to initiate them are *Nset and *Elset respectively. There's a set of sub-options used to inform Abaqus what's the name of the set and which instance it's collected from. Example of node sets being used is when boundary conditions are applied, and an example of element set is when a pressure is applied to the model.

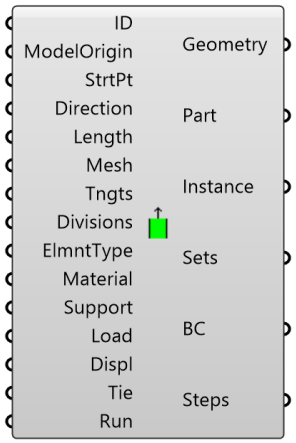
3 The Algorithm

In addition to the approach successfully discretize numerical models, it also had to be user friendly. The intention was that this approach was going to be used by others at a later time. So in order to increase user friendliness, some of the components features were given extra attention:

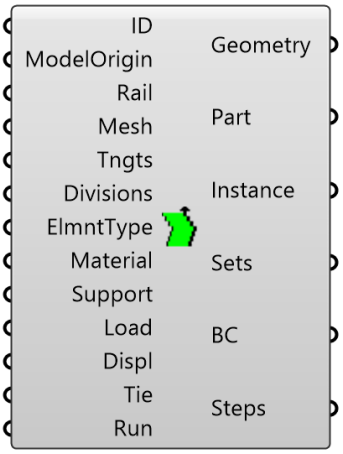
- ***Optimize calculation time:*** Calculation time is important in order to boost the interaction value. Interaction is a essential part of graphical algorithm editors such as Grasshopper. So creating components with short calculation time is preferable. Initially the code discretized the model by dividing a solid into elements. This proved to be a time costly approach since operations on solids are more CPU demanding which have a negative affect on calculation time. Knowing this, the code where reorganized so that the numerical model where wrapped around the geometry making it independent from an initial form. More details about this in section 3.1.

- ***Minimize the number of input and output:*** The information processed by the components pass through their nodes and are interlinked by wires (refer section 2.1). This means that in order for a definition to run successfully all correlated nodes on the canvas has to be connected. Connecting these nodes will become cumbersome when components requires a larger amount of input, so creating components with a minimum amount of nodes is preferable. In this algorithm, the information transferred between components had the majority of its data included in one single Data Tree. This mean when data are transferred from that component it's done with one wire, resulting in shorter assembly time of the definitions.

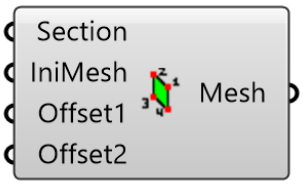
- ***Creating a natural relation between components:*** The definition where split into multiple components in order to reduce the number nodes. Due to INP files being a documentation of the pre-processor phase, the components fell in to a similar patter. The Karamba, add-on also have a pre-processor phase, so the components in this approach have a similar relation as the components in Karamba.



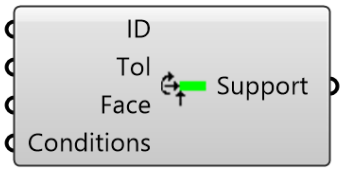
(a) Extrude Part (Appx A.1)



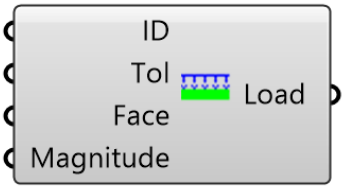
(b) Sweep Part (Appx A.2)



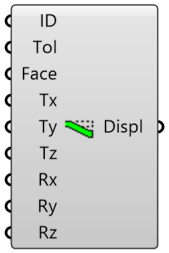
(c) Mesh Hex8 (Appx A.3)



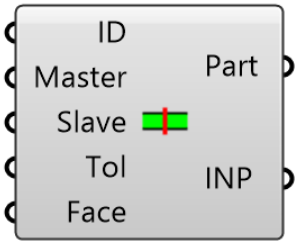
(d) Support (Appx A.4)



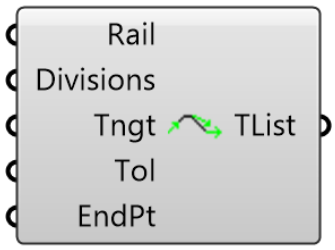
(e) Load (Appx A.5)



(f) Displacement (Appx A.6)



(g) Tie (Appx A.7)



(h) Tangents (Appx A.8)

Figure 7: Components.

3.1 Parts

In Abaqus, parts is the geometry of the model. In the INP file there is a own section for parts, where the information regarding nodal coordinates and element composition is included. In this approach, two methods for generating parts is available: extrusion and sweep (fig.7a and fig.7b). These where chosen since they're common methods for generating solids in Grasshopper. There's a toggle input ***Run***, which acts as a stream filter within the code. When the toggle is turned on, the component starts to generate the numerical model. The components are similar by the fact that they operates on curves. The difference is that the extrusion component generates a curve defined by a start point, tangent and a length (same method as the Line SDL command in Grasshopper), while the sweep gets its curve through the input node ***Rail***. This curve is divided into an number of segments equal to the input ***Divisions***, and is evaluated at the end points of each segment in order to find the nodal coordinates and the tangents of the curve. These coordinates and tangents is variables used in the transformation of the Mesh surfaces.

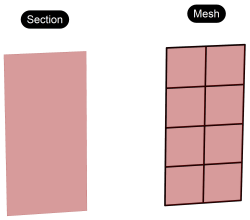
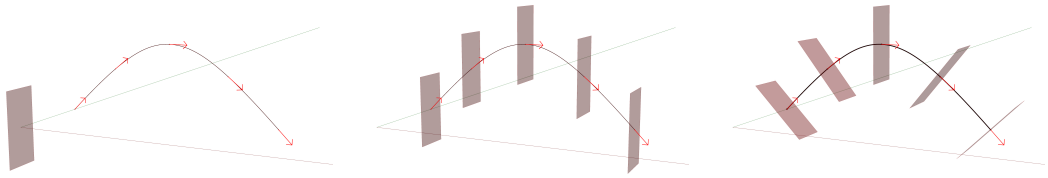


Figure 8: The cross-section and mesh of an rectangular part.

In this approach, the mesh is represented by the subdivided surface which represents the cross-section of the part (refer fig.8). The surface elements in the mesh acts as groups of nodes, where each surface vertex represents a node. Since the surface elements govern the position of its nodes, makes it possible to disretize the model by positioning the surfaces instead of each individual node. Through duplication and transformation (according to the global origin), each individual surface element is placed into position (fig.9). More details about mesh surfaces in section 3.2

The physical model displayed in Rhino has two states: before the and after discretization. The reason for dividing the geometry into two states is to



(a) Initial Position (b) Translation. (c) Rotation.

Figure 9: Transformation of surface elements.

give an visual indication if the algorithm ran or not, strengthening its interaction. Prior to the discretization (fig.10), the component places the surfaces from the mesh and the section surface in position (Duplicate ->Transform). When in position, the vertices (nodes) and edges (curves) of all surfaces are duplicated and exported to a given data tree or list. The vertices exported to the data tree *Elements* and the list *Vertices* are data used later on in the discretization of the model. The edges of the section is exported to the data tree *EdgeCurves* to be input for the geometry created when the toggle is on. It's the edges of the mesh surfaces, exported to Geometry, that's displayed in Rhino Indicating the position and orientation of the element faces. This results in a contour of the final shape. Note that the section surfaces at the end points of the curve are added to the list *Surface*, these surfaces are used to create a closed solid in when the discretizations runs.

When the toggle is on (model is being discretized), the component starts to generate the solid of the part displayed in Rhino (fig.11). The solid is built up by a series of surfaces generated by the Loft command; A loft surface is created between each point on the curve for each edge of the section . These surfaces are added then to the list *Surface*, which is joined together to a closed solid when all the edges are lofted. Since the FEA is limited to linear geometry, the shape of the sweep becomes that of the sweeps in Abaqus (linear curves between nodes). Figure 12 shows the two states of display.

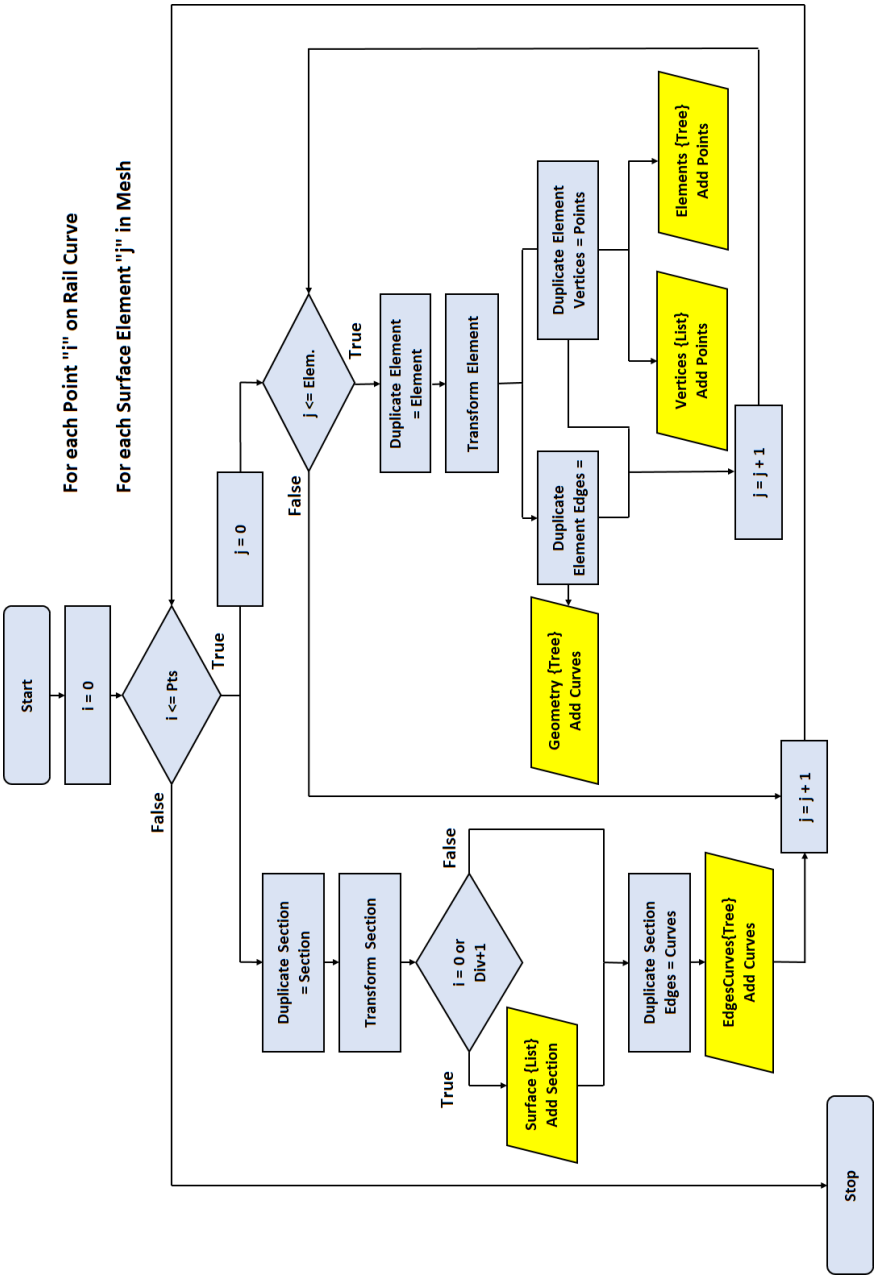


Figure 10: Algorithm unaffected by RUN.

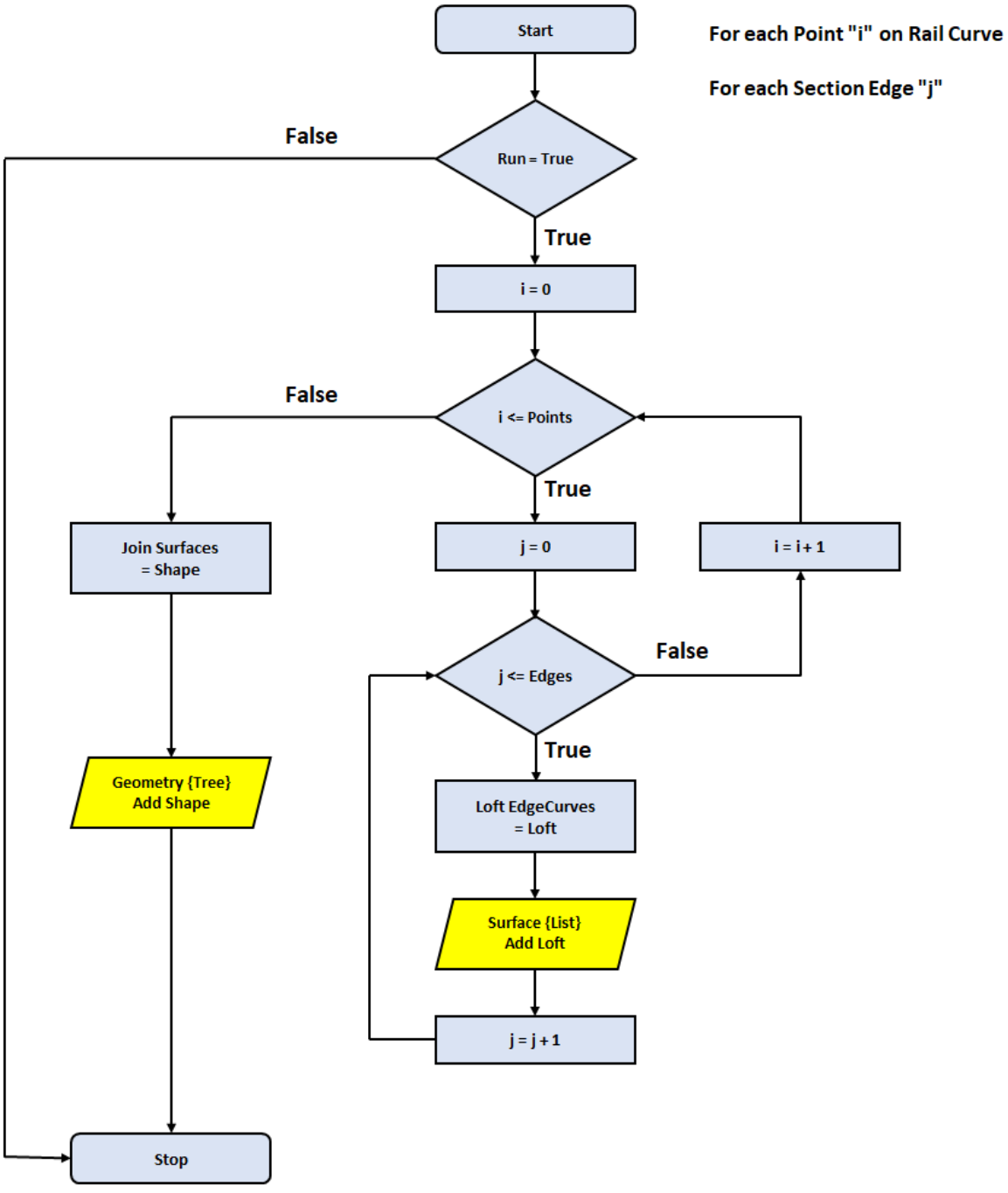


Figure 11: Part of algorithm that generates brep dependent on Run.

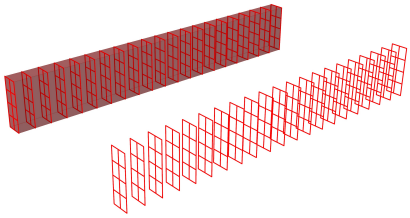
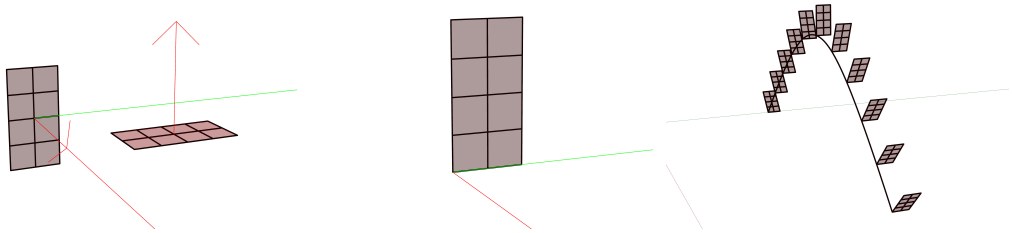


Figure 12: Generated geometry.

3.2 The Mesh

As mention is previous section, the mesh in this approach is a subdivided surface. The approach is derived so that the mesh can be defined at any position in the project, giving the possibility to define a section from an arbitrary geometry in the project. In order for the discretization to run successfully, the mesh surfaces have to be managed in order to meet some criteria. These operations are performed by the Mesh Hex8 component (fig.7c).

The first thing the component does is to reorient the section and the mesh. Surfaces are translated so that the centroid of the section is positioned at the global origin, and then it's rotated so that the surface normal is aligned with the global x-axis (fig.13a). This gives a consistency not only for the transformation process to come but also for the rebuilding of elements the elements. Note that the component has the inputs ***Offset1*** and ***Offset2***, these inputs allows the user to move the mesh so that the rail curve used by the part generators doesn't align with the section centroid. This is useful when the given rail isn't the center line of the section but rather one of its corners (fig.13).

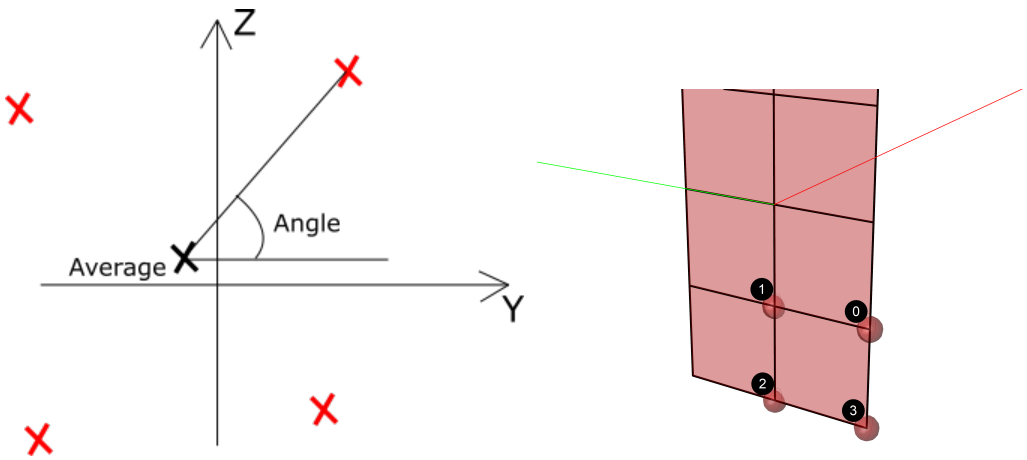


(a) Reorientation of Mesh. (b) Mesh is offset. (c) Surfaces moved to rail.

Figure 13: Reorientation of Mesh.

Since the mesh can be any divided surface, given that each subdivision has four vertices, there's no certainty that the vertices will be ordered in a structured way. This means that when the nodes of the part are added to the *Elements* branch (refer section 3.1) they are ordered randomly, which leads to a problem when the elements are discretized. Section 2.3 showed that the elements shape depends on the topological mapping of the element. To ensure that the element gets the correct mapping, the Mesh Hex8 component rebuilds the surfaces to ensure a consistent listing of the elements.

To rebuild a element, the component extracts its vertices and calculates the average point of these. This average point is used as origin point to calculate the angle between it and each individual vertices. After sorting the vertices by its angle to the average point, the surfaces are rebuilt so that the indices of the vertices are consistent (fig.18). This ensure that the element in the part components have consistent connectivity.



(a) Calculate angle. (b) Vertices order.

Figure 14: Rebuild surface element for consistent vertices order.

3.3 Discretize the physical model

When the toggle input ***Run*** is on, the part components starts to generate the numerical model. From the way the INP file is structured, the components starts by listing the nodal coordinates. The list *Vertices*, which includes all the vertices of the surface elements in the mesh, is the basis in defining the parts nodal coordinates (fig.15). Since the nodes are placed into position using surface elements, the vertices on adjacent surfaces overlap creating duplicate points. If the duplicates is not removed from the list, then Abaqus will use all points threatening the continuity of the mesh. In the case of duplicates, the nodes with the same nodal coordinates are managed as individual nodes. That means if one of the node is displaced, the other won't follow. In order to deal with this issue, the duplicate points in *Vertices* are removed by calculating the distance between each node in the list and removing those who have a length close to zero. This results in the new list *Nodes*.

After the duplicates are removed, the code starts to add the nodal coordinates to the *Part* list. This list contains the data for the part used in the PARTS section of the INP file. In the INP file, the declaring of nodes are initiated by the option Node (refer section 2.3) which means that the keyword *Node is added to the list before the nodal coordinates. Note that the part components have the input ***ModelOrigin***. This option is used to define the origin point for the model in the Abaqus/CAE interface. If it's not specified, the INP file uses the global origin. When the exported geometry is located far away from the origin, it becomes hard to navigate in the Abaqus/CAE model. Before the algorithm appends the nodal coordinates to the list, it checks if components have an input value for the model origin. If there's an input, the nodes from the list is subtracted the input value before it's appended to the list. If not, the nodal coordinates are appended to the list directly.

After the nodal coordinates are appended to the list, the code starts to add the elements topological mapping to the INP file. The keyword *Elements with the sub-option type equal to the input ***ElmntType*** is added to the *Part* list to initiate the declaring of elements. The element type, tells Abaqus which kind of integration method[4] is used in the calculation and adapts the elements accordingly. The available integration methods in this approach are: Full integration, Reduced integration and Incompatible modes (fig.17). In determining the mapping of each elements, the algorithm uses the data tree *Elements* which is structured so that the vertices of each individual mesh surface at a given position on the curve has it own path. At

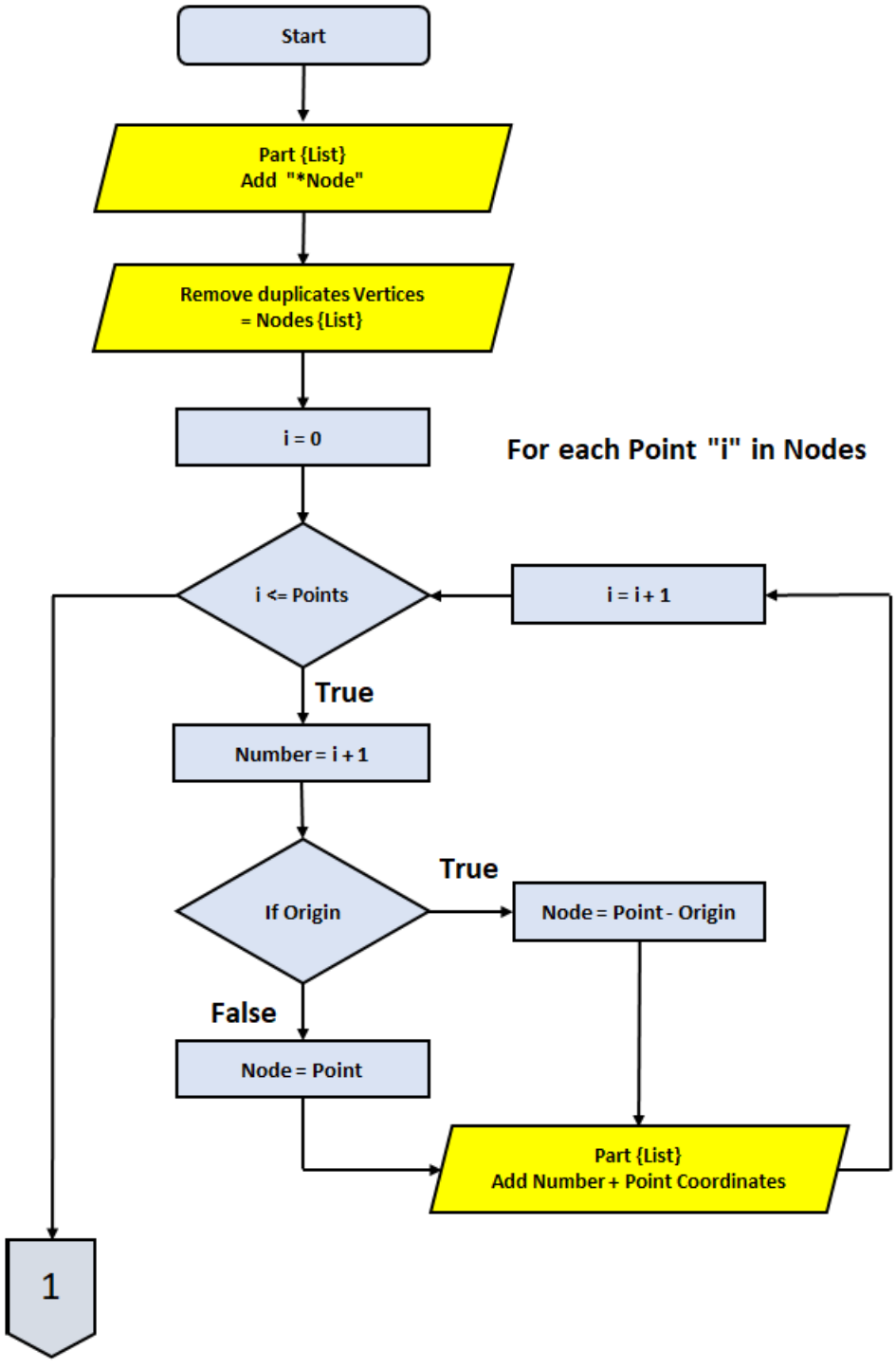


Figure 15: Discretizing the Nodes of the physical model.

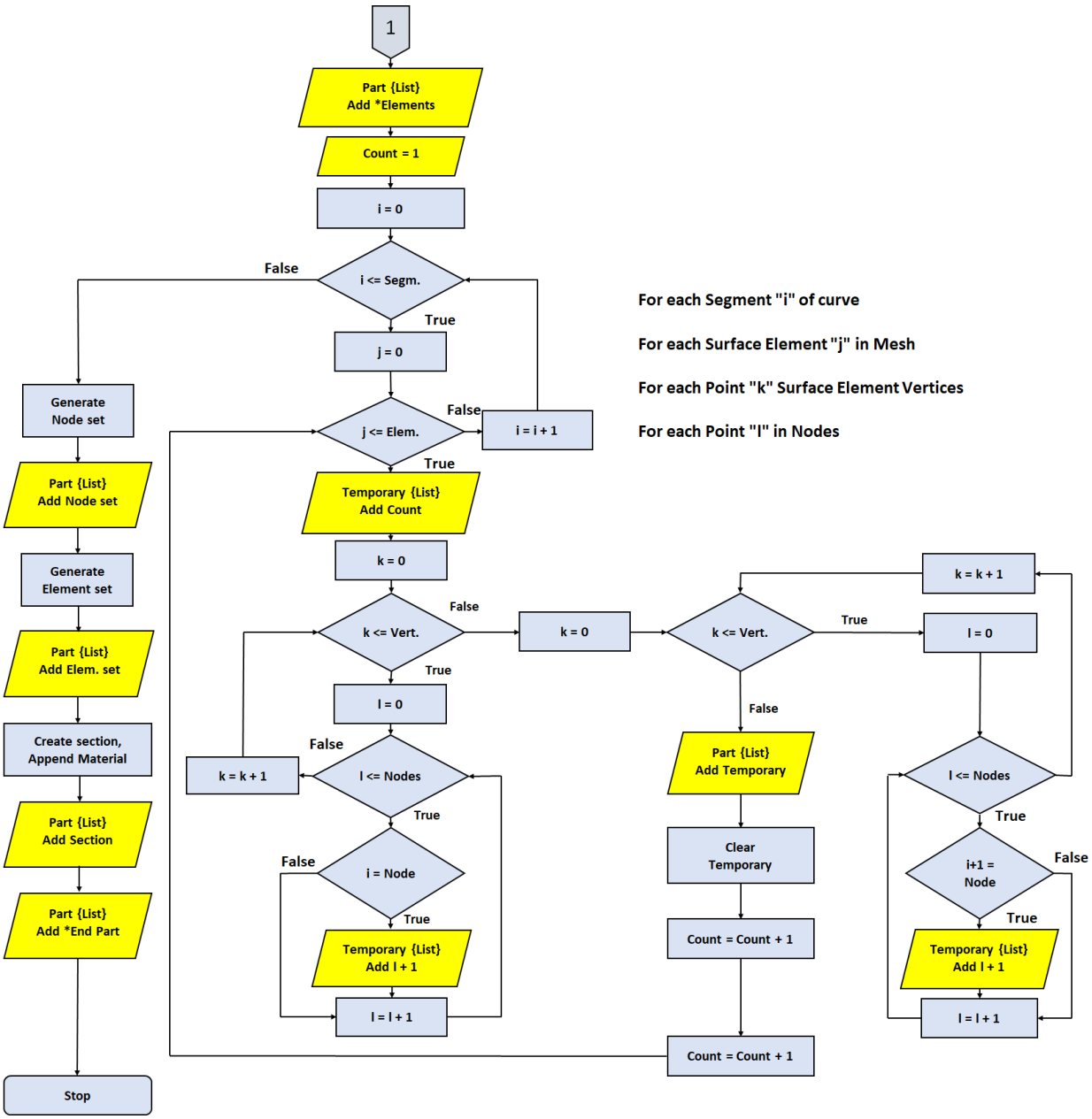


Figure 16: Discretizing the Elements of the physical model.

each segment of the curve (element) the algorithm compares the points of the element faces S1 and S2 (fig.18a) to each node from the *Nodes* list to find the node indices of the element corners. In this scenario, the surface element at the start of the line segment represents S1 and the surface element at the end represents by S2 (fig.18b).

After all the element in the part is identified, the algorithm create a set of all the nodes and a set for all the elements. These sets are added to *Part*. Before the part is finalized, the algortihm create a section which uses the newly created element set and assigns it the material properties (fig.19). This command set the material properties of the elements. To finish the discretization, the command *End Part is appended to the list.

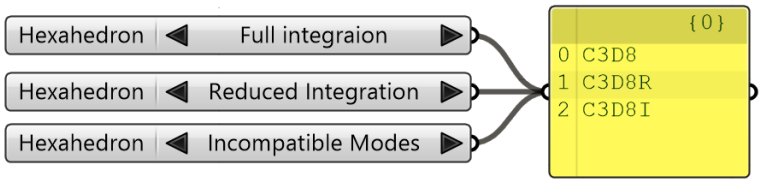
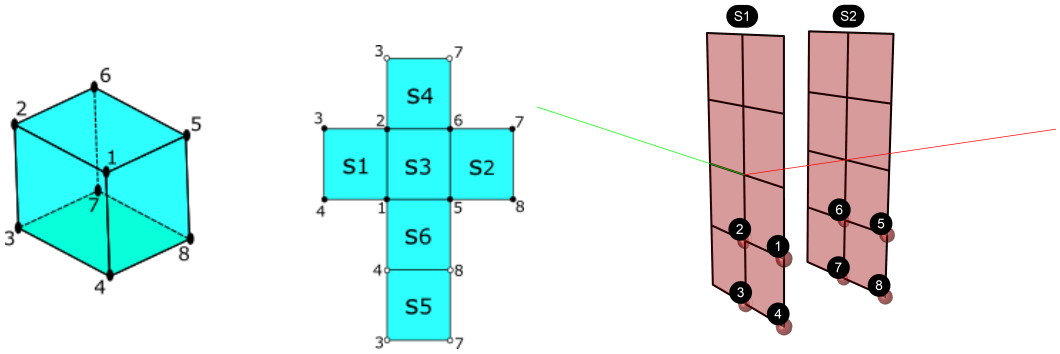


Figure 17: Integration method available in approach.



(a) Connectivity hexahedron element. (b) Vertices order.

Figure 18: Discretization of elements.

Figure 19: Component for adding material properties.

3.4 Performing operations

Abaqus performs actions on the model by assigning an action onto a set (refer section 2.3). In the INP file, an action is initiated by a keyword which calls a set to operate on in its definition. To create sets, the algorithm uses surfaces. The surface is placed on the model and the components register which of its nodes that's on that surface (fig.20). To identify which nodes that's on the surface, the node is projected onto the surface and then the distance between it and its projection is calculated. If the distance is within a certain tolerance, a procedure is initiated depending on which kind of set the surface is identifying (surface or node set).

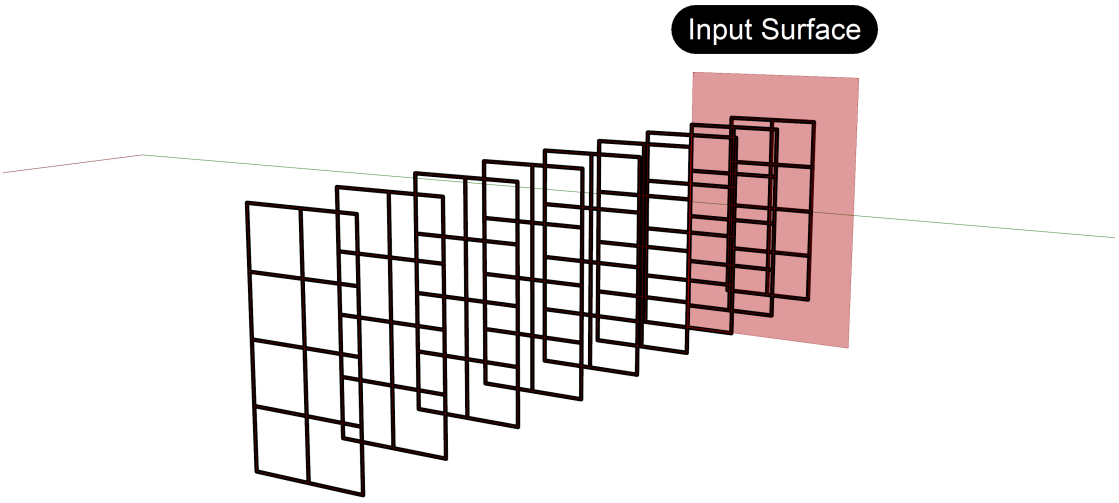


Figure 20: Input surface used for generating sets.

When creating node sets, the component adds the index of the node on the surface to the set directly. While for identifying surfaces, the component have to identify 4 nodes (equal to the number of vertices of the element face) before it adds the face of the element to its set (fig.21). In order to add the correct face to the set (fig.18a), the component runs through all the elements in the part. For each point on the surface, it adds the index the node has in its element topological mapping, to a temporary list which resets between elements. If that list gets 4 indices, the current element is added to the set and its side is identified. The side is identified by comparing the indices in the list with the topological mapping; If the list contains the indices $\left[5,6,7,8\right]$ the side is identified as S2, for $\left[1,2,5,6\right]$ it's S3 and so on.

The components in the algorithm that uses surfaces to define sets are: Support (fig.7d), Uniform Load (fig.7e), Displacement (figure 7f) and Tie (figure 7g). The support and displacement components generates node sets whereas the uniform load and tie generate sets of elements. It's worth noticing that the Tie component have an extra output. This is due to the fact that this component ties part together, a tie is an interaction module that couples surfaces from separate parts together during the simulation [11]. The surfaces in the tie are assigned roles, one as *Master* and the other as *Slave*. The slave surface adjust itself according to the master, therefore it's recommended that the section with highest mesh density is assigned as slave. Since the tie component demands input from two separate parts, it generates the input for the INP assembler directly (refer section 3.6). Figure 22 shows how ties are set in this approach.

For very refined meshes, the distances between nodal points can become so short that the input-surfaces register the points on the surface and those close to it. To counter this a tolerance input ***Tol*** is included. This input lets the user adjust the sensitivity the algorithm uses when determining if a point is on the surface. Also, in the case that the surface isn't positioned close enough to the nodal points, the sensitivity can be adjusted so that it register points further away.

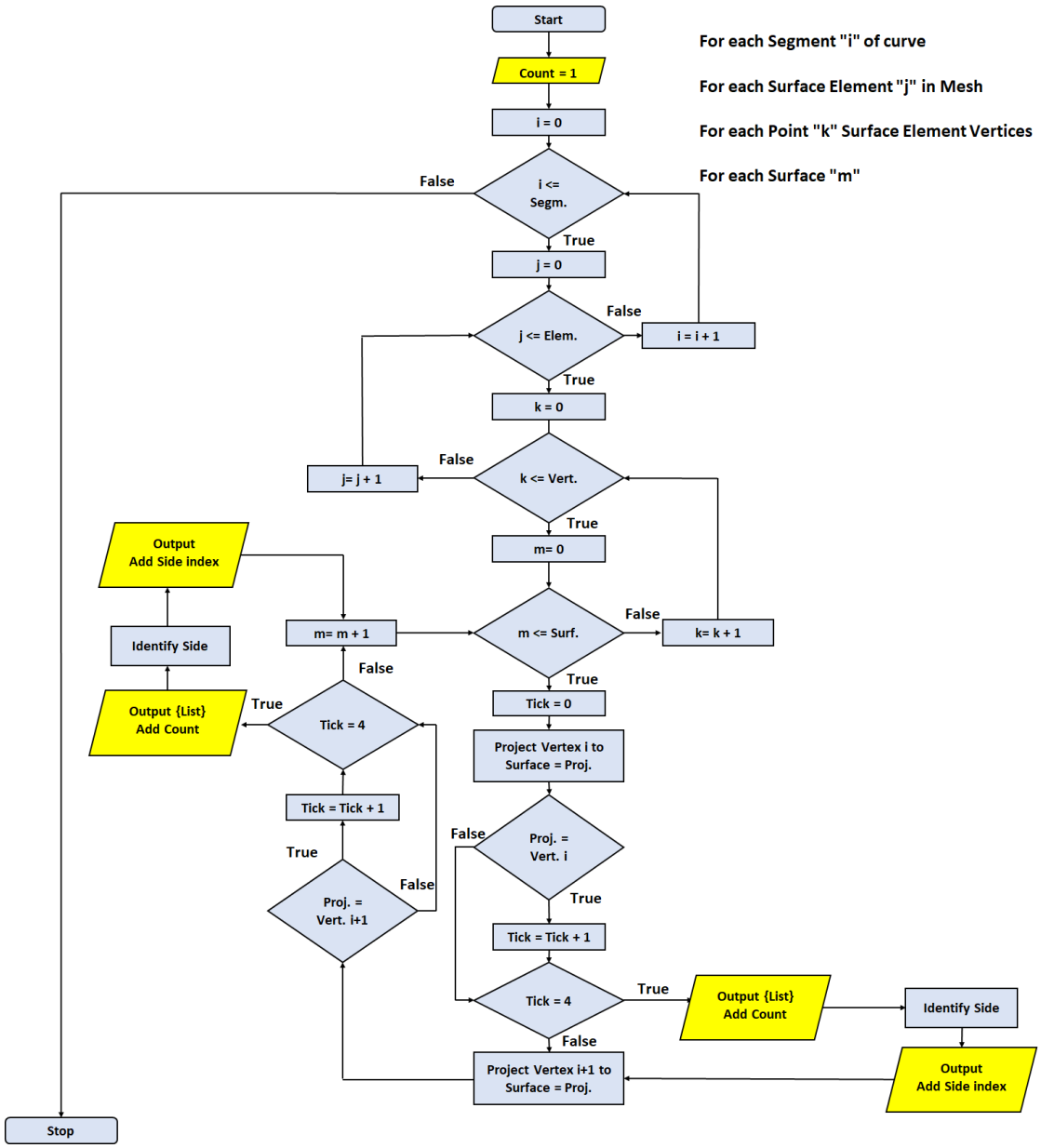


Figure 21: Method for identifying surfaces.

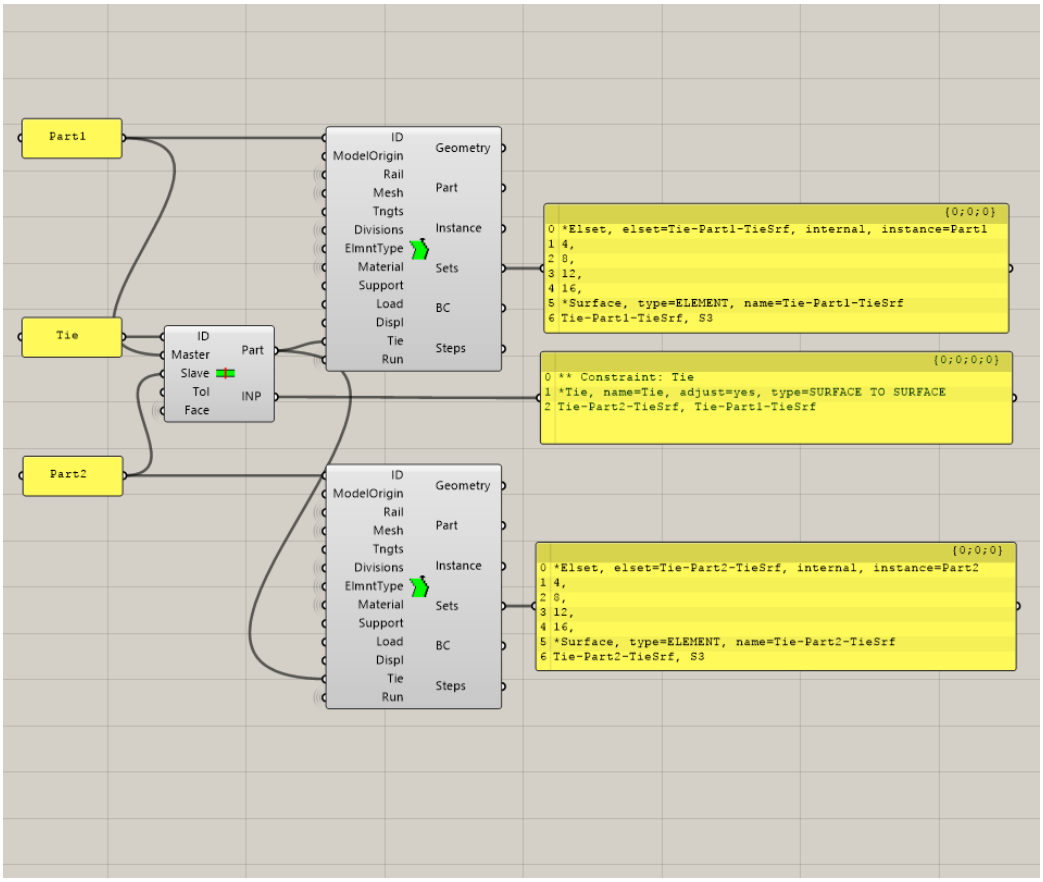


Figure 22: Two parts tied together.

3.5 Skewed geometry

To account for skewed geometry, which is common for connections, the part components have an input ***Tngt***; A list of vectors, that overrides the rotation of the mesh surfaces so that their surface normal aligns with the input. The first vector in the list overrides the tangent at the start of the curve, the second on the tangent at the first division point, this goes on until the list ends or there's no point left on the curve to override. Each time the mesh surface is rotated according to an input vector the height of the surface is elongated. To account for the elongation, a ratio for scaling the height of the surface is derived. Examining a skewed element face (fig.23) one can derive a scaling factor by looking at the ratio between the initial surface height and the elongated height.

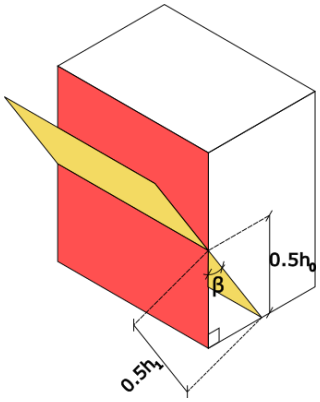


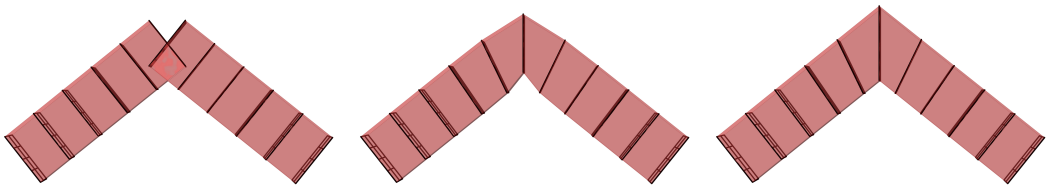
Figure 23: Skewed element face

$$\cos(\beta) = \frac{0.5h_0}{0.5h_1}$$

$$h_1 = \frac{h_0}{\cos(\beta)}$$

$$Scalingfactor = \frac{h_1}{h_0} = \frac{h_0}{h_0 \cos(\beta)} = \frac{1}{\cos(\beta)}$$

Using this relation to scale the surface element height prevents the geometry from getting kinks along its sides (figure 24b). Scaling of the mesh surfaces is performed during the initial transformation process (refer section 3.1). By including skewed geometry, this approach is capable of covering more designs.



(a) Initial orientation (b) Unscaled (c) Adjusted

Figure 24: Skewed connection

The vectors that the input ***Tngt*** receives, can either be defined manually or by the the component *tangent list* (fig.25). Tangent list is a component which generates a list of vectors by comparing the override vector with the

curves tangent. The component start by adding the override vector ***Tngt*** to the output list, then it calculates the mean vector between the input vector, the tangent it overrides and the succeeding tangent. This average vector is also added to the list. Then a certain condition has to be met for each succeeding point on the curve:

$$1 - |V_{avg} \cdot Tangent_{succeeding}| < Tolerance$$

If this condition isn't met, then the average vector is recalculated using the previous average vector, the curve tangent at the current point and at the succeeding tangent. This continues until the condition is met, or until the end of the curve. When the tolerance is met the component returns the tangent of the curve at that point and all succeeding points. Since the component operates on unit vectors, the difference between 1 and the dot product will converge towards 0 as the average vector closes in to the succeeding tangent

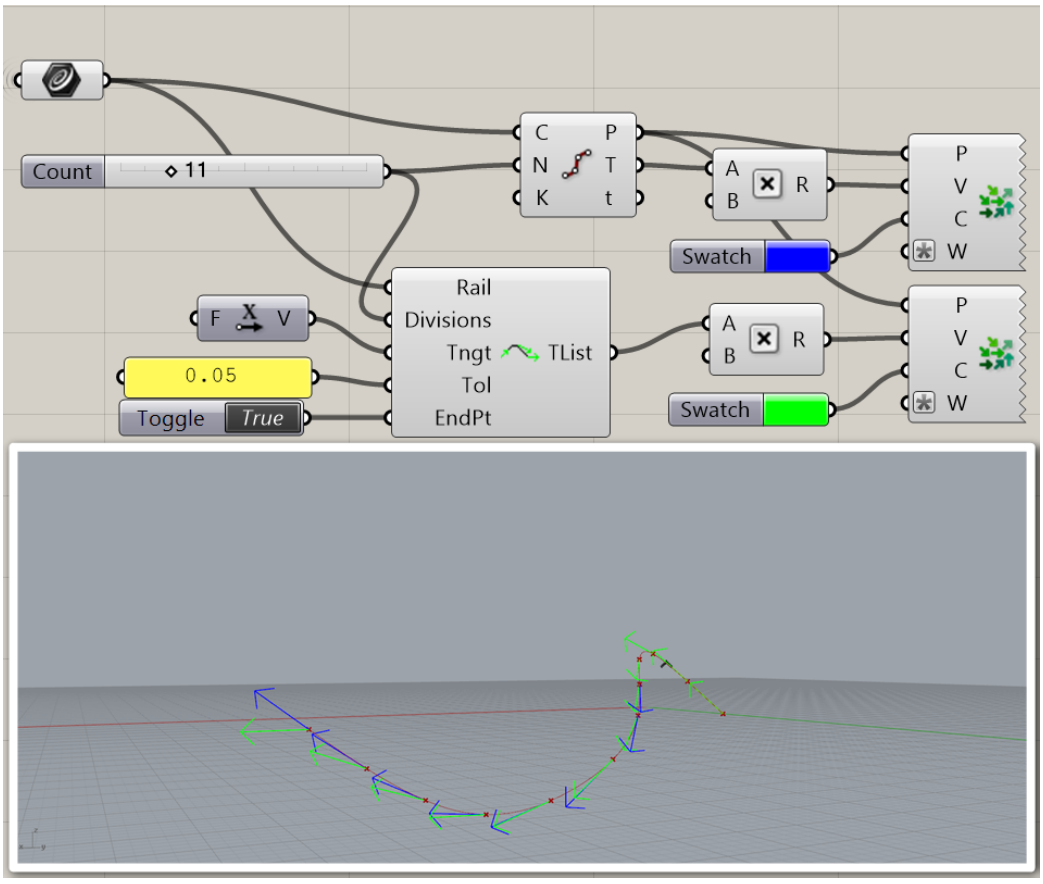


Figure 25: A tangent list definition

This component has its limitation, when working with refined meshes and the override vector might make the elements in the mesh overlapping creating discontinuity. In these situations it's best to manually define the vector list.

3.6 Assembling the INP

In order to export the model, all the information from the discretization needs to be collected and arranged into one document. This is what the INP Assembly component does (fig.26). The code of the component contains all the sections of the model, e.g. **PARTS, **ASSEMBLY, **MATERIAL, etc. The input is collected and added to their respective section in the file. When the model have multiple parts, the input nodes of the component has to be flattened in order for the information to be arranged correctly. Connecting the output from the INP Assembly to a panel makes it possible to stream the content of the INP file directly to a text file. Right clicking the panel gives access to the stream options. When the stream option is active, Grasshopper does real time update on the document at the stream destination. NB! In order to import the text file to Abaqus the filename extension has to be inp.

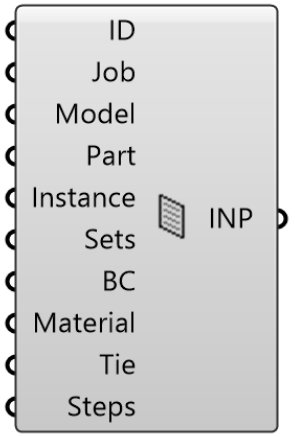


Figure 26: The INP assembler component (Appx A.9)

4 Case Study

To demonstrate the functionality of the approach, the upcoming sections presents a series of cases which the approach has been applied.

4.1 Cantilever Beam

In order to show how to assemble a definition, this section present a simple example using a cantilever beam. This is a classic statically problem frequently used in many textbook problems and tutorials, which makes it a suitable case to present how to define the code.

The definition starts by determining how the beam will be created. In this case the beam will be defined using the *Sweep Part* component, which means that a curve is created as the path of the model. Figure 27 shows the definition of a curve, its length is governed by the slider *Length*. When the value of Length changes so does the length of the line and also the length of the beam.

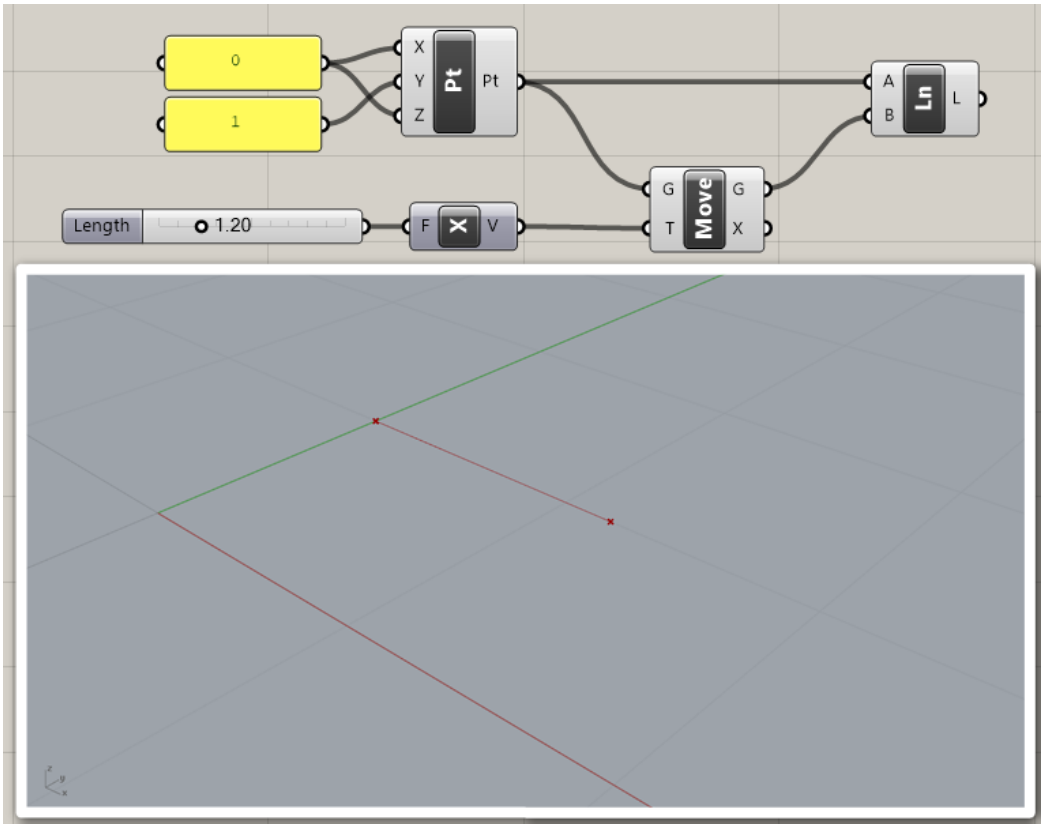


Figure 27: Curve defining the path of the beam

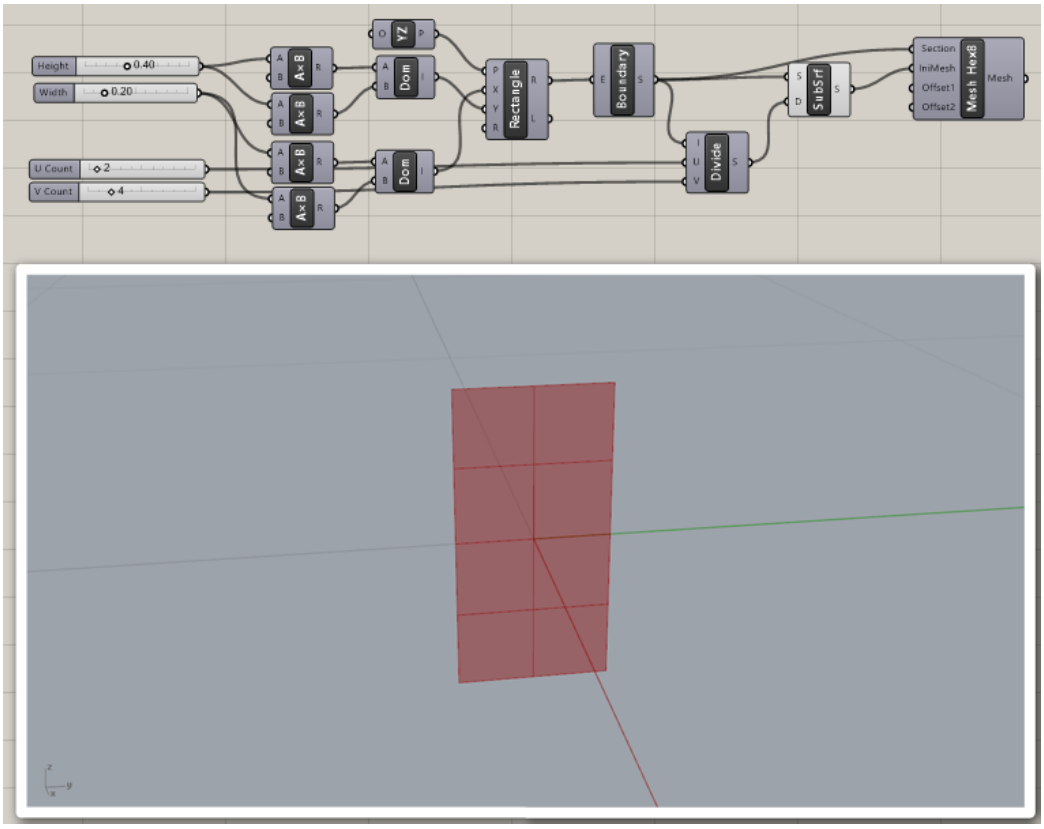


Figure 28: Mesh of a simple rectangular beam

In order to show how to define a mesh, a simple rectangular beam is used as a cross-section. Figure 28 shows the mesh used in this example. The surface representing the cross-section is subdivided using a Grasshopper component called isotrim, it extracts an isoparametric subset of the surface defined by the divide domain component which divides the surface domain according to its input U and V Count. Increasing the value of these will refine the mesh.

With a rail and a mesh defined, it's possible to generate the contour of the shape (represented by the edges of the surfaces in the mesh). Connecting these and an integer slider for the Divisions to the sweep component generates the contour shape of the beam. The window in figure 29 displays the contour of the beam. The definition in the figure includes a series of additional input: Material properties, identification of the part, element type and the toggle. These inputs are not dependent on input from other components,

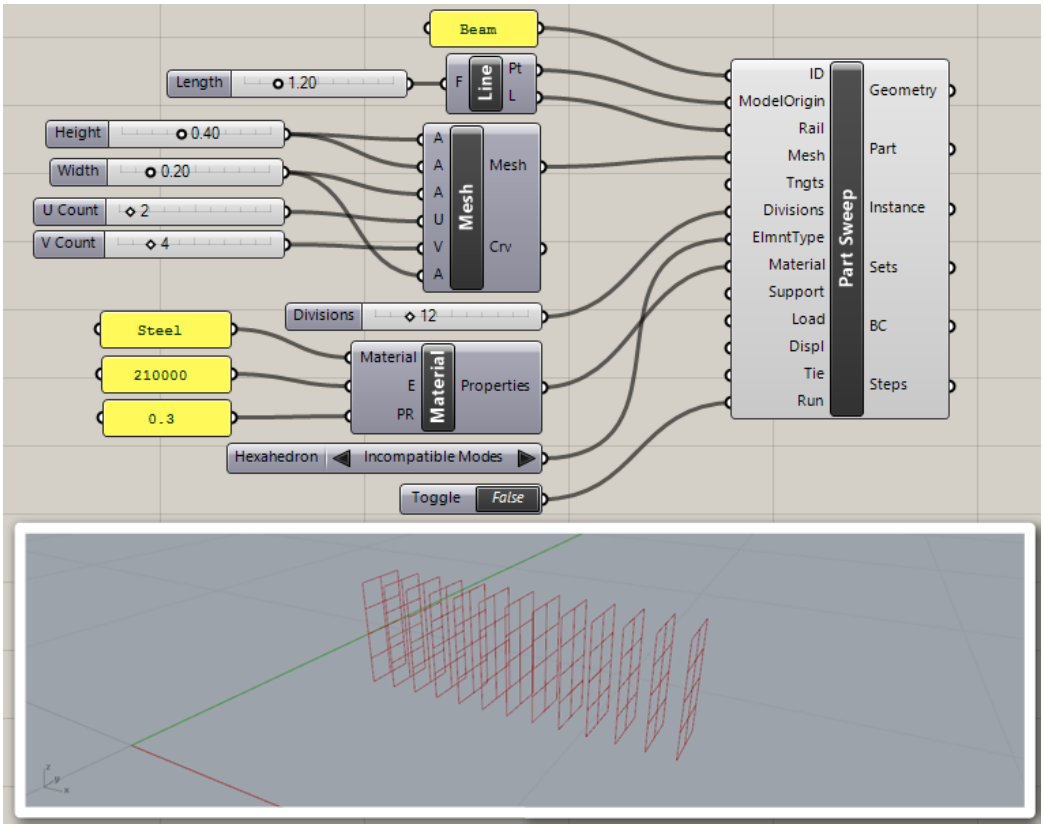


Figure 29: Connect components to display contour of beam

except for the material properties but it is declared using panels which is easily appended. Note that the model origin is connected to a point from the line, this is to improve navigation in the Abaqus/CAE window.

The restraints are assigned to the beam using surfaces. The Plane Surface command in Grasshopper, is a convenient way to create surfaces used by the support component. By creating a plane on the curve defined by one point on the curve and the tangent at that point, that plane can be used by the Plane Surface component to create a surface on that plane. In this definition, the curve is evaluated at the start of the curve to find the coordinates and tangent. A surface is constructed at the start of the curve by using this data. This surface is connected to the support component input Face. NB! It's important that the dimensions of the surface is greater than the section in order to identify all restrained nodes.
The restraints are called using lists, if the list includes [1,2,3] the nodes

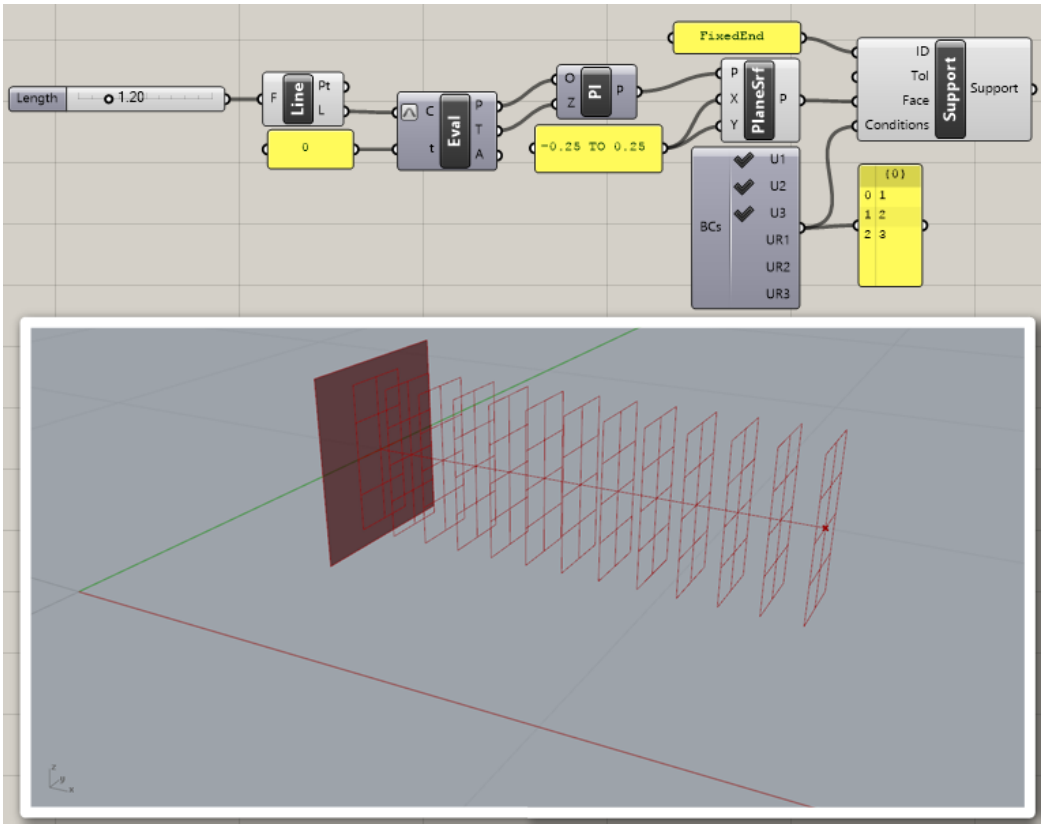


Figure 30: Applying Boundary Conditions

on the surface are restrained against translations. The tick off menu in the figure generates a list with integers depending on which variables that ticked off. The component also register strings *U1*, *U2* and *U3* when calling the restraints. Six DOF are available for restraint even though there's only three DOF are available for solids, they're included to be applied on other element types in future versions.

Similarly as to the support, the load is assigned using a surface. Here, the upper edge of the section is copied and swept along the rail resulting in a surface used to define a pressure load at the top of the beam. This surface is connected to the uniform load component and it applies a pressure to the faces adjacent to the surface.

Figure 32 shows the final definition of the cantilever beam. To create the INP file, an INP Assembly component is placed on the canvas and it's connected

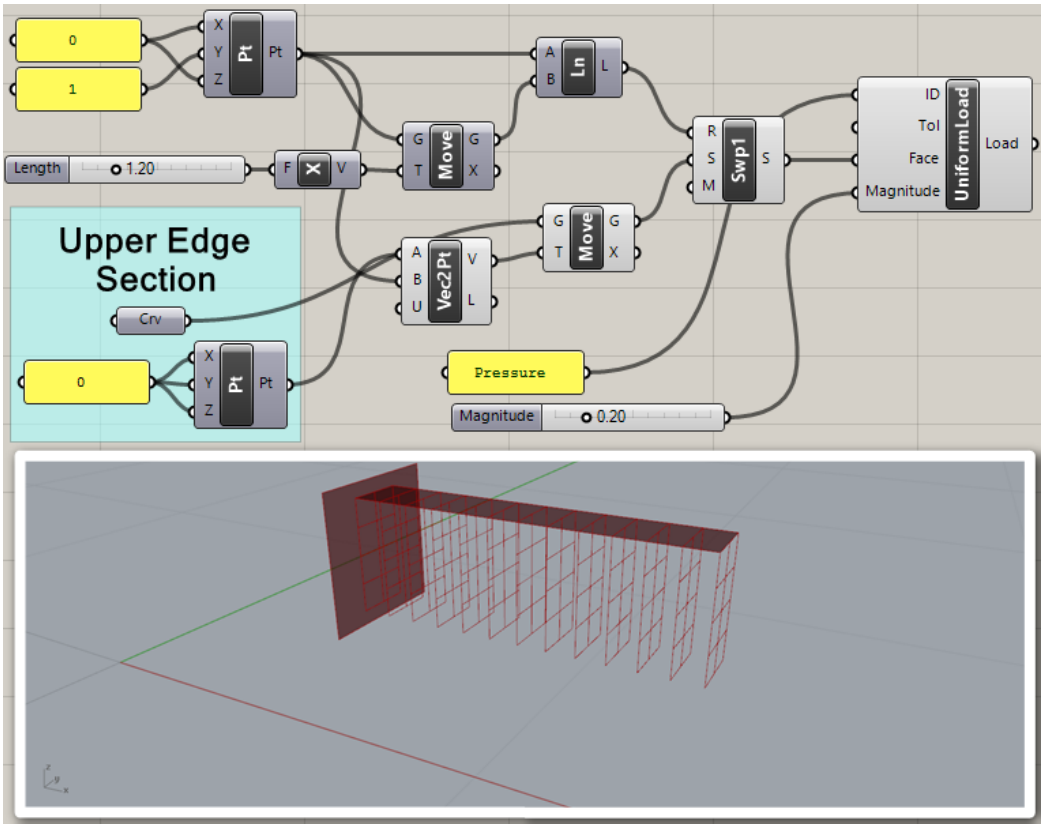


Figure 31: Applying Pressure Load

to the model information. Note that the material properties component is connected to the assembler, this is so that the INP file can declare material properties. These properties are not a output from the part component due to the fact this material can be connected to multiple parts creating a series of multiple output. The INP file output is connected to a panel in order to stream the numerical model to the .inp file.

The model is then imported to Abaqus and a simulation is run, the result is shown in figure 33. This simulation shows that there is a connection between Grasshopper and Abaqus.

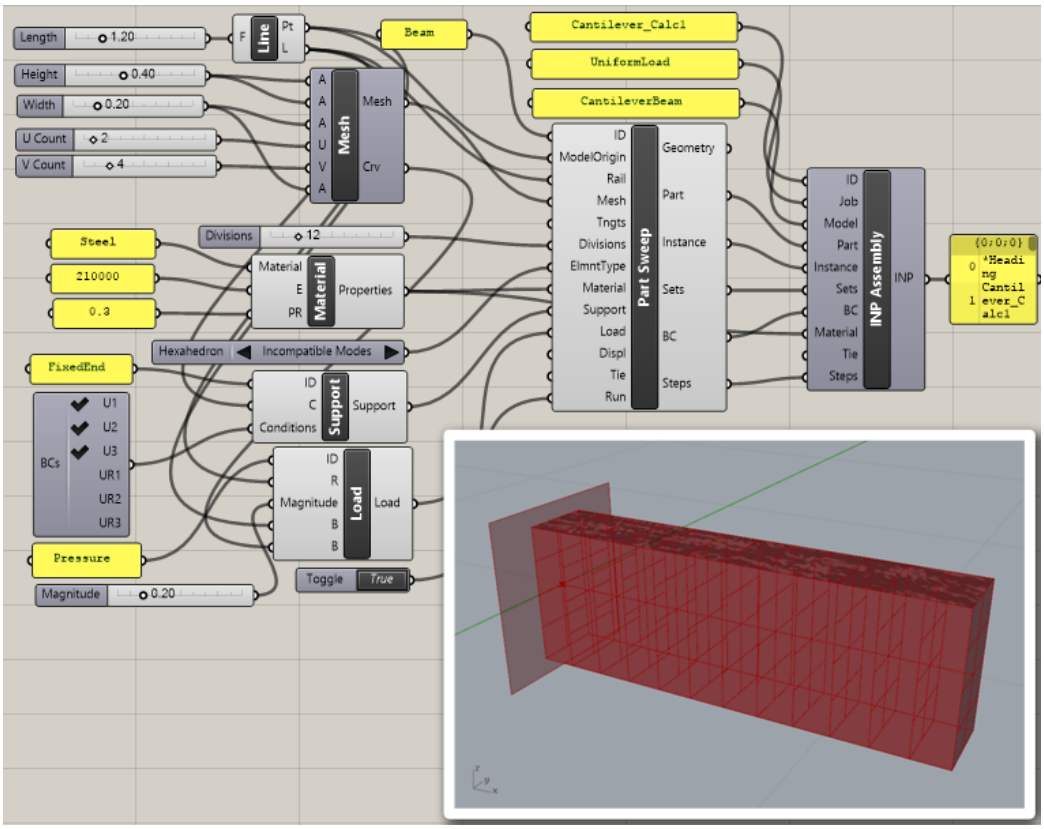


Figure 32: Finalized Definition

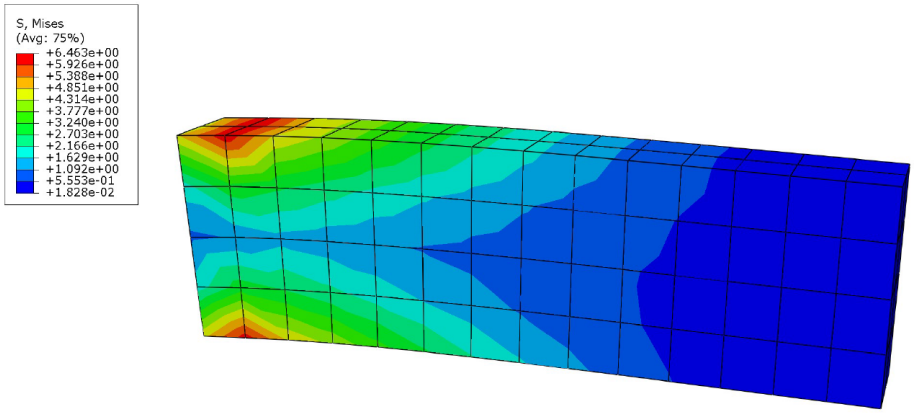


Figure 33: Result from simulation

4.1.1 Mesh Sensitivity

A short mesh sensitivity study is performed in order to demonstrate how the parametric in the model work. In this definition, the mesh is governed by the inputs *U Count*, *V Count* and *Divisions* (fig.32). Higher the value, more refined the mesh gets. A beam with the dimensions: 1.2m length, 0.4m cross sectional height and 0.2m width, was applied to 0.5MPa; Four instances with cubic elements were analyzed, with refinement of 2^2 for each instance (fig.34).

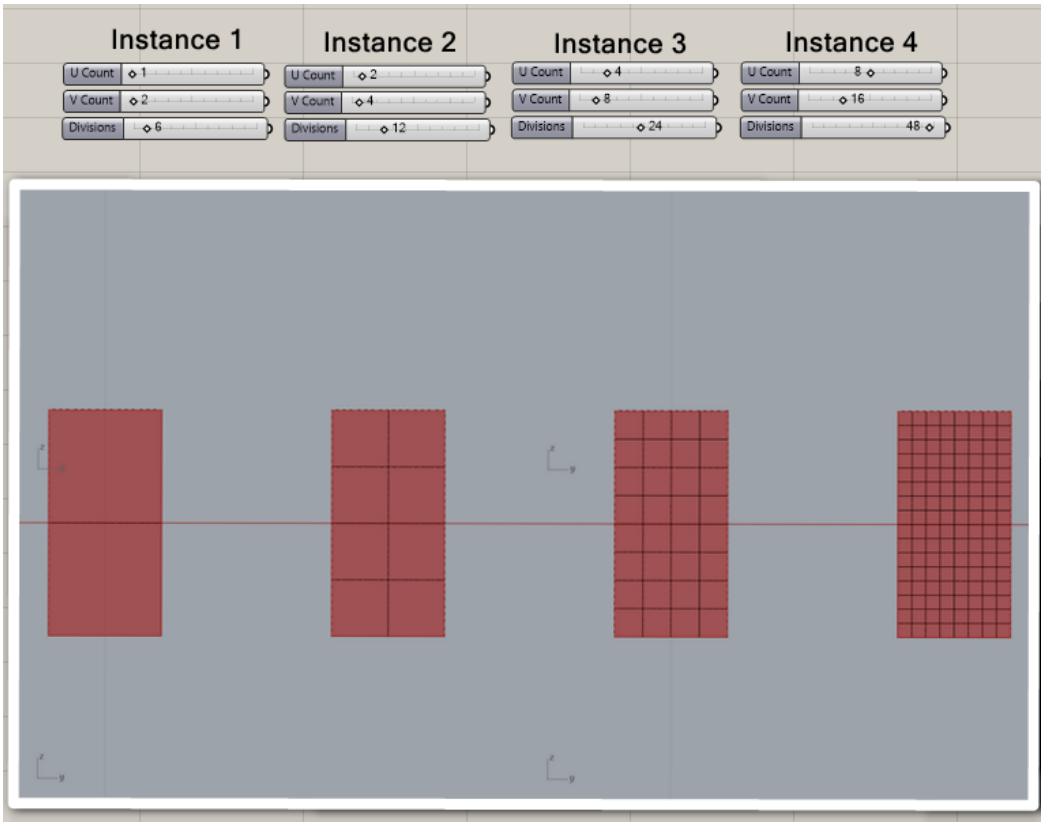
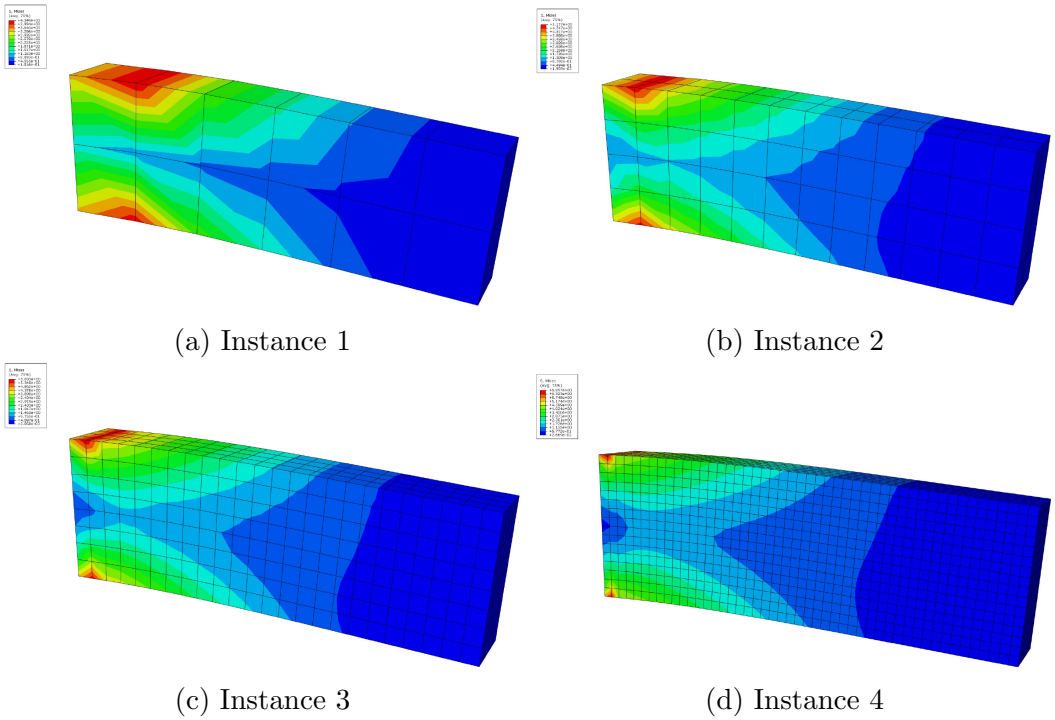


Figure 34: Cross section of instances.

Due to the theory of elasticity and the non-physical (fixed support) boundary condition, the singularity appear on the bottom and top surface of the beam. As shown in the graph (fig36), the maximum stress in concentration place is exponentially converging to the unrealistic big value.



(a) Instance 1

(b) Instance 2

(c) Instance 3

(d) Instance 4

Figure 35: Analyzes results for the different instances.

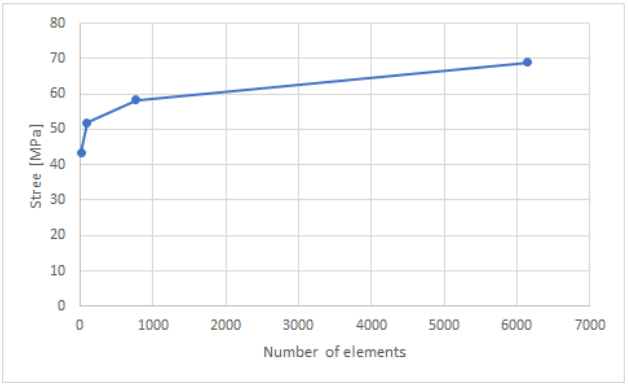


Figure 36: Results of results mesh refinement

4.2 Direct connection Karamba

To examine the possibility to include output data from Karamba, a static problem with an built in beam (fig.37) was analyzed by Karamba in order to find the maximum displacement. This value was connected to the displacement component (fig.39) which integrated it to the FEA. In order for the component to identify and create a node set to displace(refer section 3.4), a surface was constructed at the point of maximum displacement.

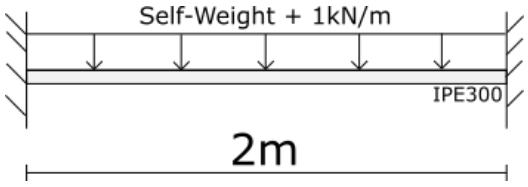


Figure 37: Statical Problem

In order to recreate the IPE section that Karamba uses in its analyzes, a component that generates mesh of IPE sections was created (Appx A.9). This component was structured so that it registers the same commands as the section component in Karamba.

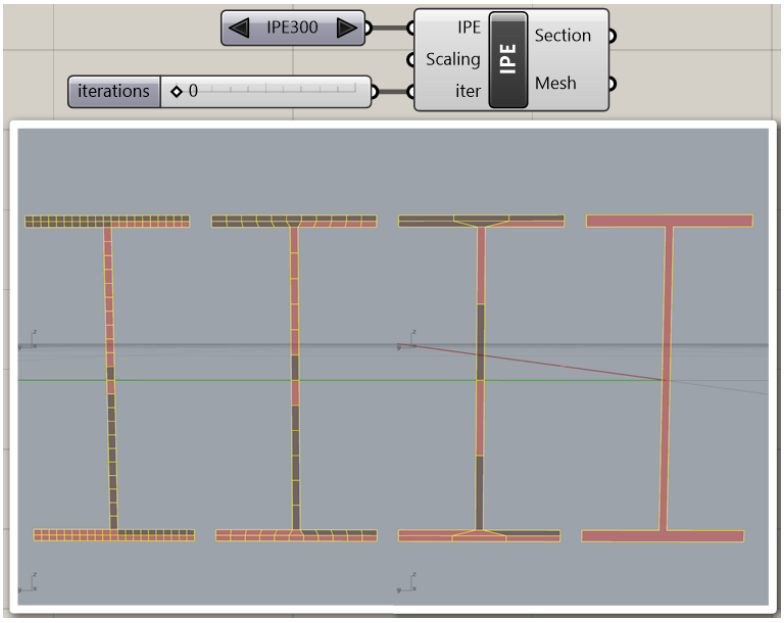


Figure 38: Component that generates IPE mesh

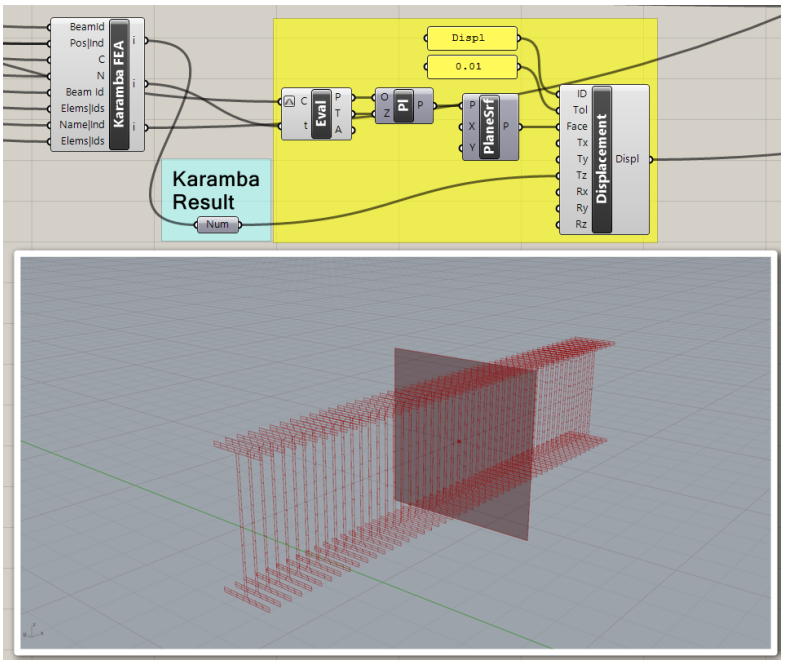


Figure 39: Karamba result connected to INP file

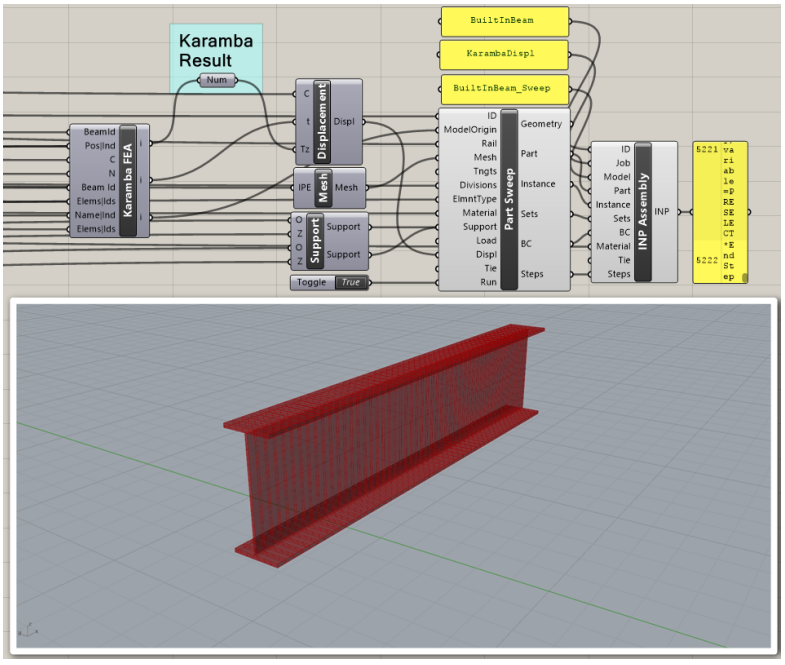


Figure 40: Results are added to the INP document

This gives the opportunity to connect both analyzes to the same input value, ensuring that the same section is used for both analyzes. The IPE components input ***iter*** refines the mesh according to its value. Figure 38 shows different meshes generated at different values of iter, values from right to left are: 0, 1, 5 and 10.

After the Karamba results and the displacement component is connected, the rest of the model is connected in a similar matter as in the cantilever case (fig.40). The definition is finalized and the FEA results are connected to the translation in z-directioin (fig.40).

The simulation from Abaqus (fig.41) shows that the displacement from the Karamba FEA have successfully been integrated to the numerical model, and thus a connection between the two programs is confirmed. This direct connection between the code and Karamba gives a real-time data-input that's applied to the FEA, boosting the level of interactivity off the approach.

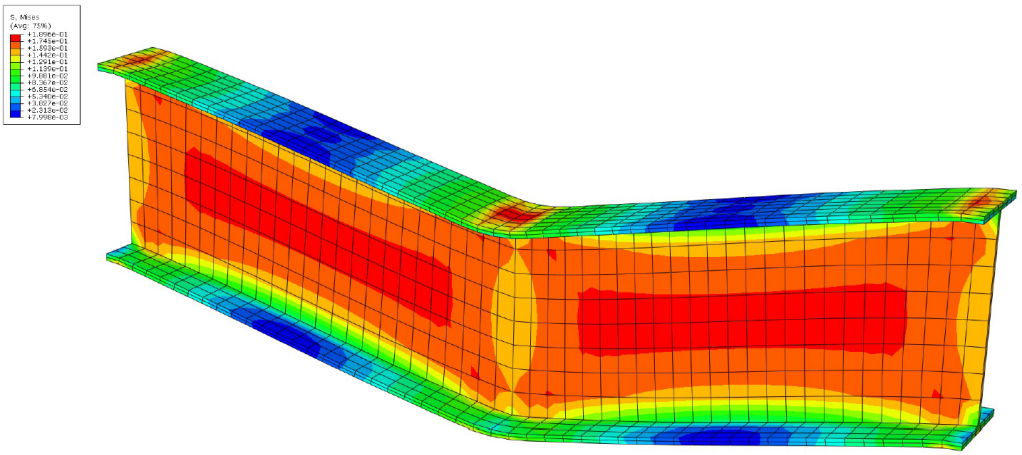


Figure 41: Result from simulation

4.2.1 Local Mesh Refinement

In this problem, the critical stresses are located at mid-span of the beam. The mesh used in the analyzes was defined globally, making all the elements in the mesh of equal in size. In order to make the simulation run more effective, the mesh should be locally refined at the critical stress region. For this approach, the local refinement were achieved by dividing the curve into three equally sized segments. Where each individual segment was the rail of one sweep, resulting into a total of three parts (fig.42).

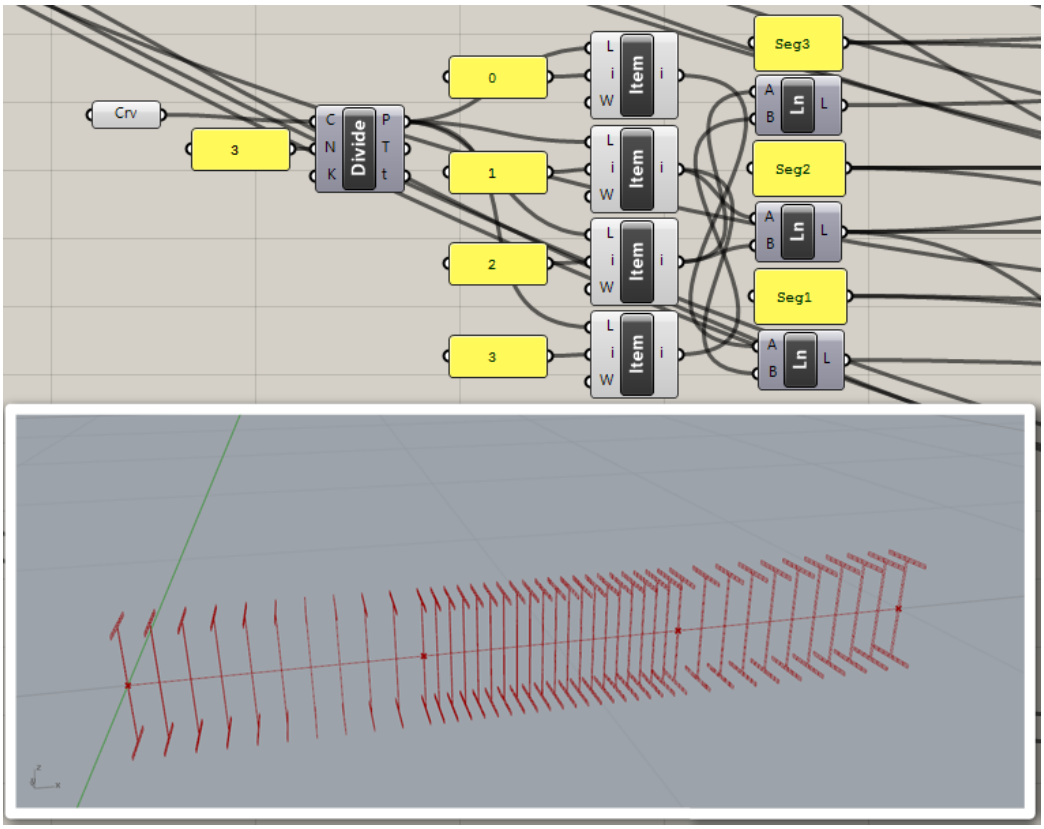


Figure 42: Beam in three parts

The parts are assigned their respective boundary conditions and connected to one of two sliders *Divisions Sides* and *Divisons Mid* (fig.43). The sliders gives the possibility to define numbers of elements in the critical region separately from the rest of the beam. The parts are then connected by coupling the adjacent faces using the tie component, in this problem it doesn't matter which part that's assigned the role of *Master* or *Slave* since their cross sectional mesh is the same (refer section 3.4).

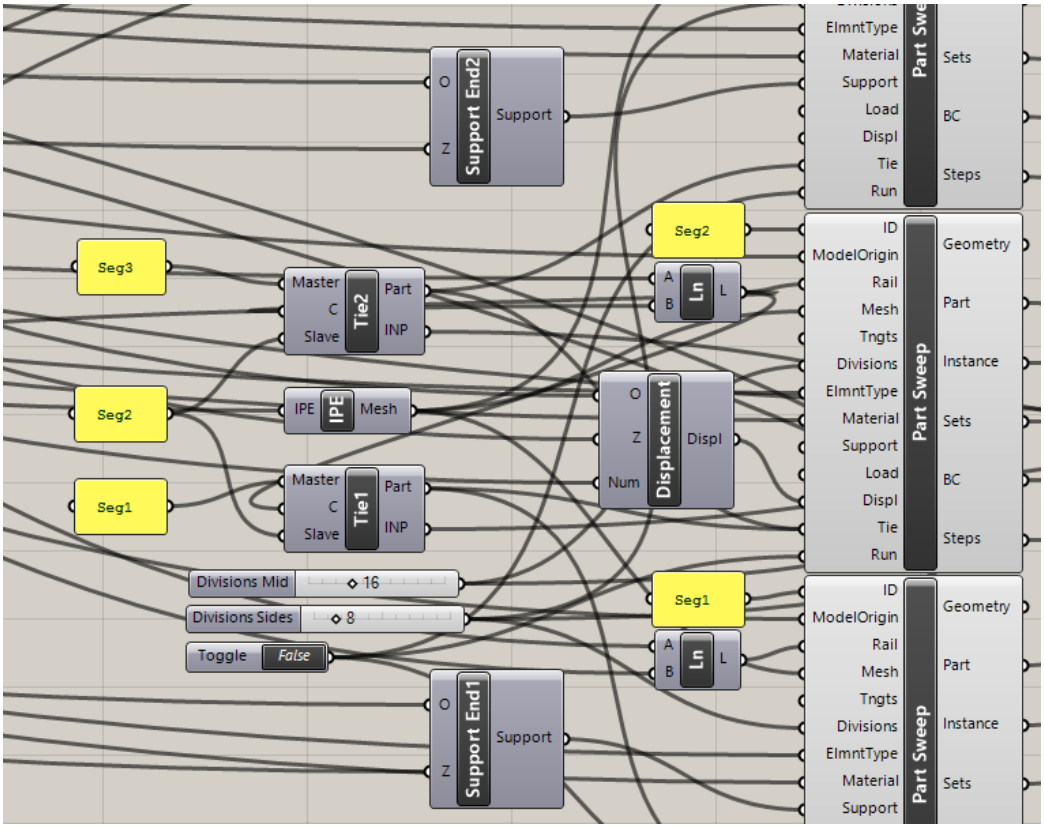
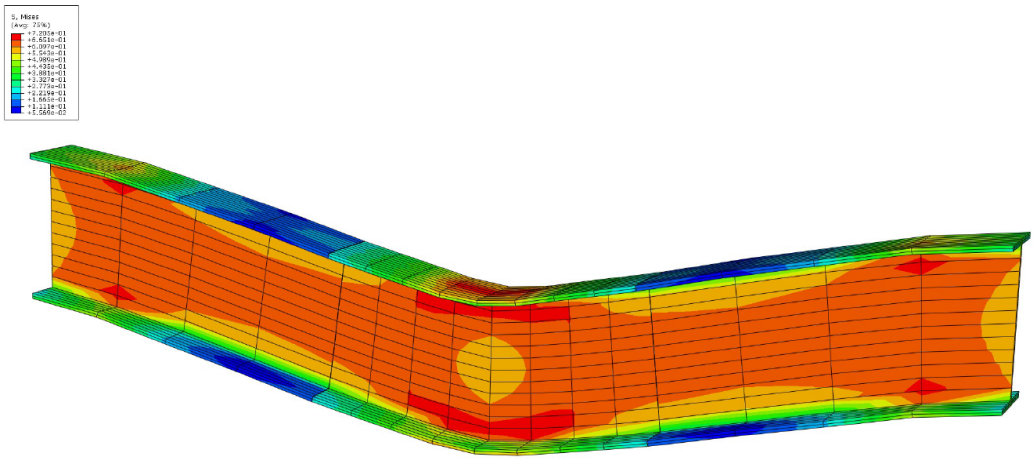
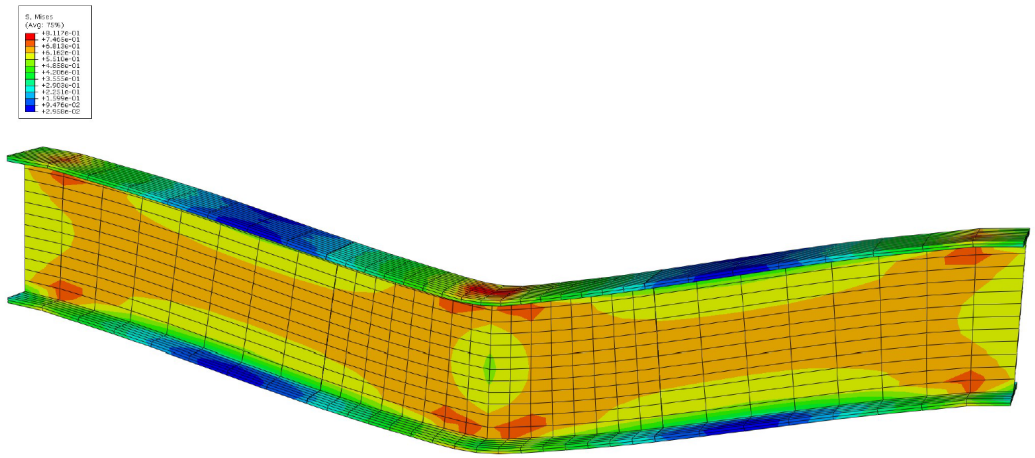


Figure 43: Definition of the model

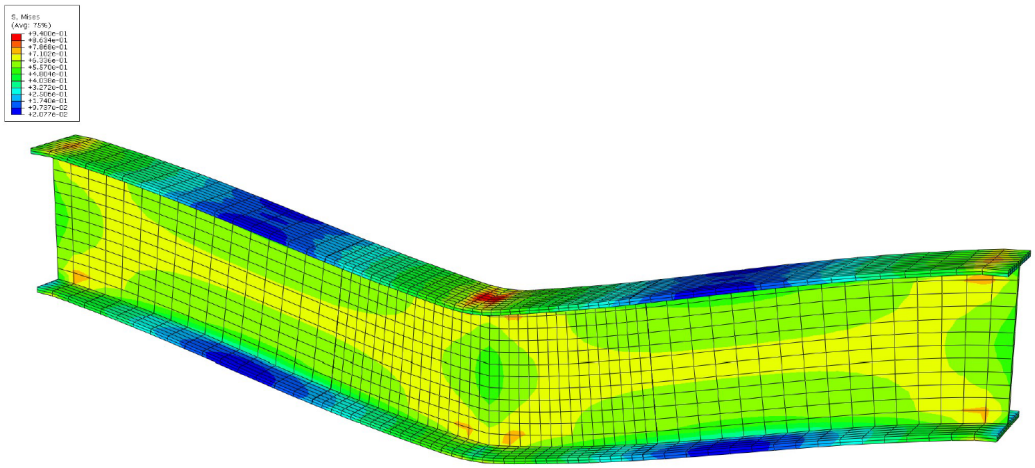
The results shows that as the mesh at mid span region becomes more refined, the critical stresses gets concentrated more locally thus increasing the accuracy refer figure 44. This demonstrates that it's possible to use this method to create locally refined mesh.



(a) Coarse Mesh



(b) Refined Mesh



(c) Fine Mesh

Figure 44: Analyzes results for different grade of meshes.

4.3 Steel Connection

When working on simple steel connections, the design is based on the EC3-1-8. In order to implement the requirements from EC3-1-8, predefined solutions must be created since the code operates on basic components, Per K. Larsen [8]. The components in the predefined design consists of basic components from the code creating a possibility to connect EC3 to the approach. To examine this possibility, a beam to beam connection was modeled (fig.45).

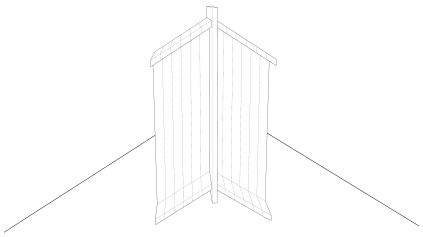


Figure 45: Beam to beam connection

Creating a parametric definition for the connection required several geometrical operations and components (fig.46). Recreating or managing this definition requires insight its procedures and structure, which doesn't make it as flexible as it could have been. A predefined solution should be easily implemented and its data should be few and understandable. To make the definition on the canvas predefined, a new component was created that compiled the information on the canvas into a single module (fig.47). The input required from a project is a line and one of its end points. From this input, the components generates the connection about the point according to the input meshes (one for the plate and one for the beams). To save input and calculation time, the component only creates model for FEA with deflections.

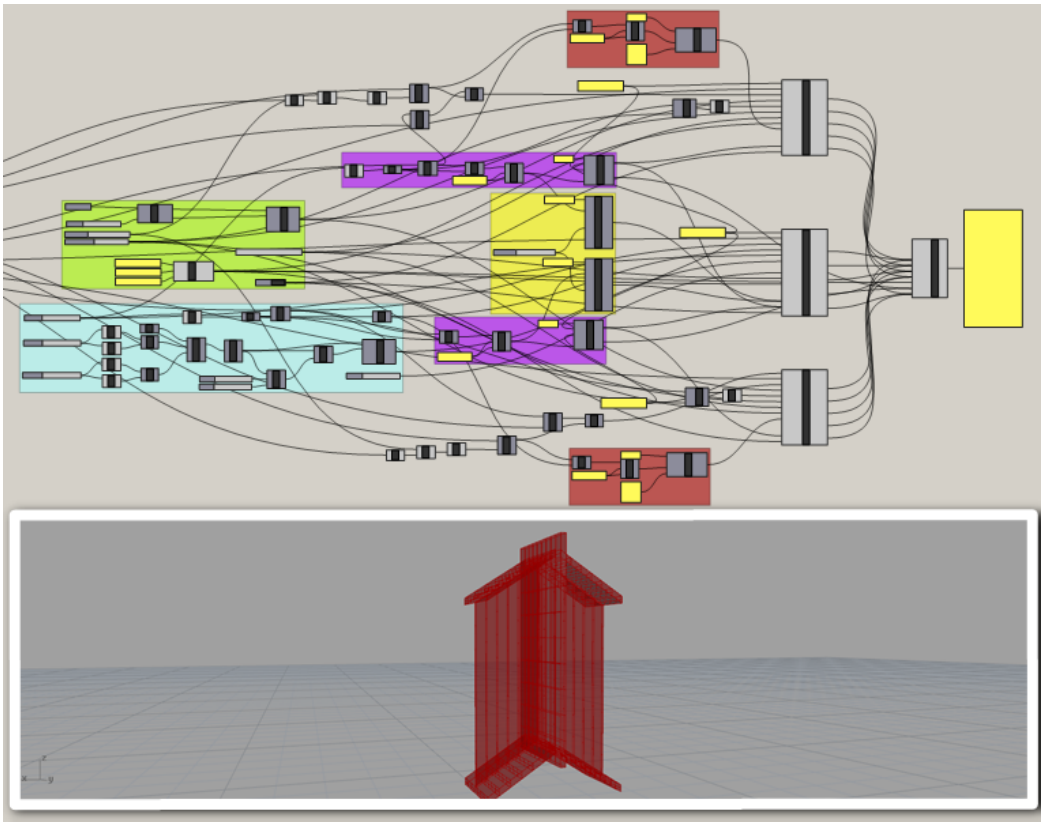


Figure 46: Definition of the joint in Grasshopper

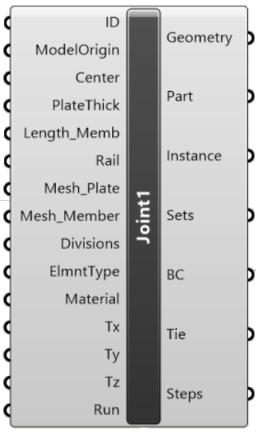


Figure 47: Component of the joint (Appx A.10)

4.3.1 Simulation

In this section some different simulations will be presented to show that the approach is capable of exporting numerical models to FEA. It is assumed that the weld is the strongest component in the connection.

The first simulation (fig48) shows the result of a simulation where a truss with a 45° pitch is deflected with 1mm at mid span. The second simulation (fig49) shows how the mesh density affects the stress distribution. This model is also deflected 1mm at the mid span. The results (fig.48) shows that the critical

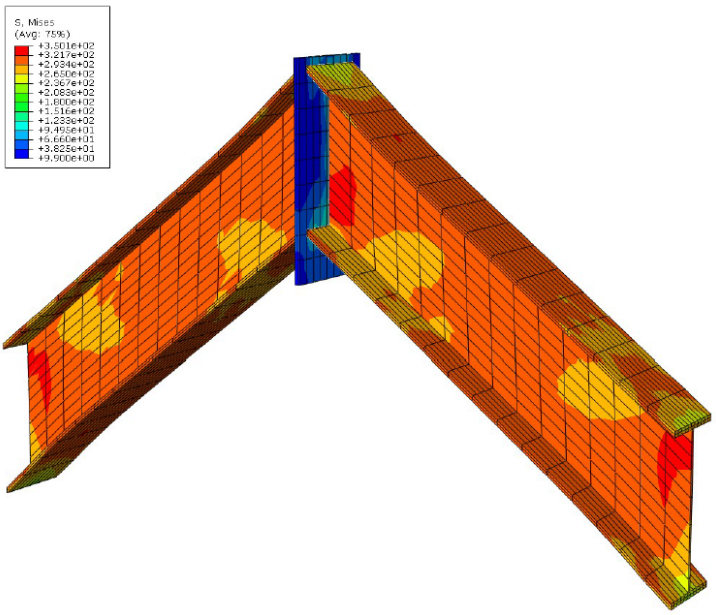
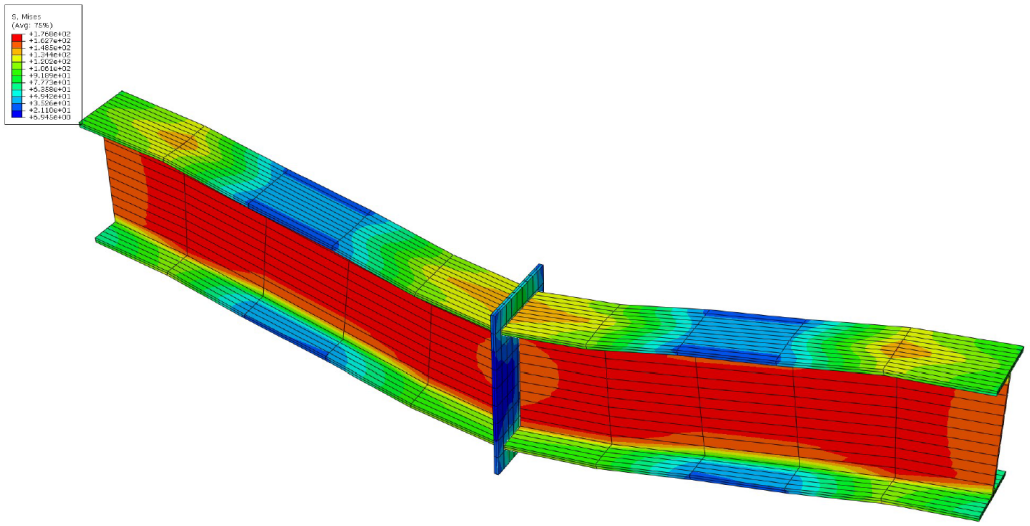
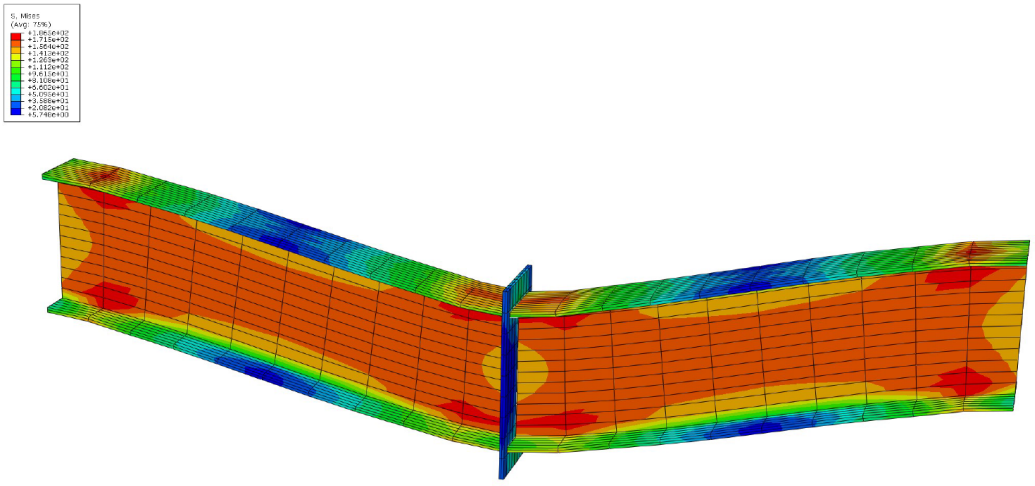


Figure 48: Analyzes of connection

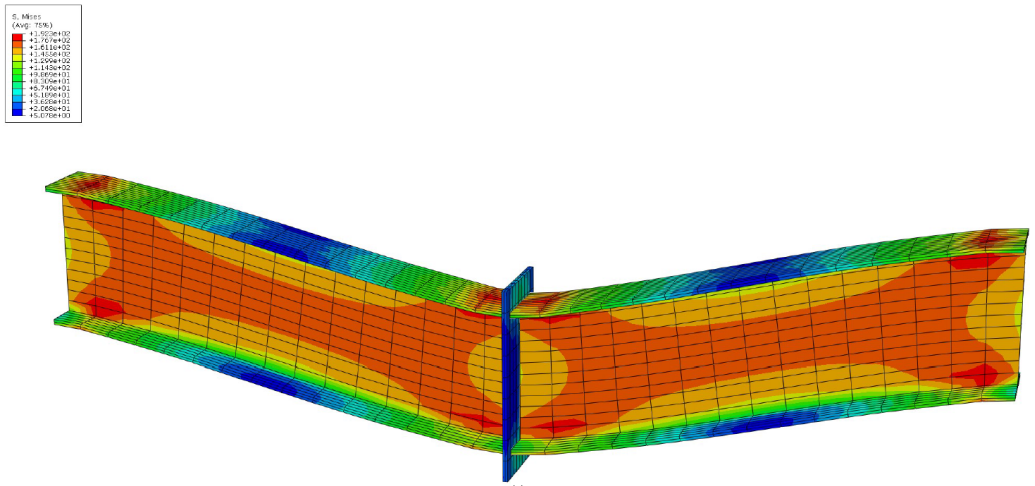
stresses are located at the base of the beams and at the connection. While the results (fig.49) shows that as the mesh gets more refined, the critical stresses becomes more concentrated, similar to the refinement from section 4.2.1. These results shows that it's possible to create and export numerical models from Grasshopper to Abaqus.



(a) Coarse Mesh



(b) Refined Mesh

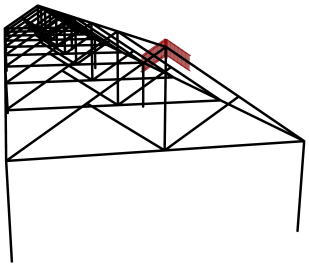


(c) Fine Mesh

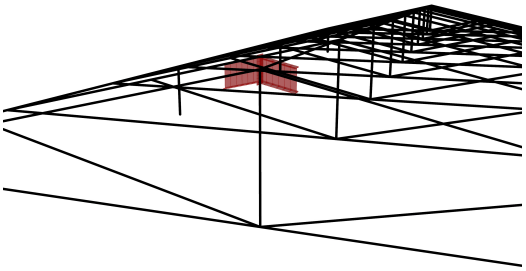
Figure 49: Analyzes results of refined mesh, connection.

4.3.2 The components fexibility

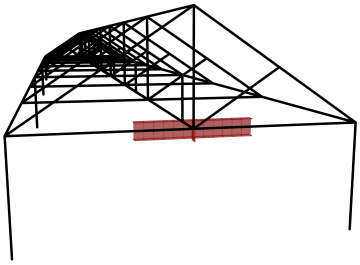
The module for the predefined connection was derived so that it could adapt itself according the geometry given. Figure 50 shows how the connection follows the slope of a gable truss as the dimensions of the truss changes. This demonstrates how a fully parametric connection behaves, showing the potential of this approach. When the INP file assembler is running, it is constantly changing the numerical model according to the connection (fig.51), and it is possible at any point to export the numerical model to FEA.



(a) High inclination angle

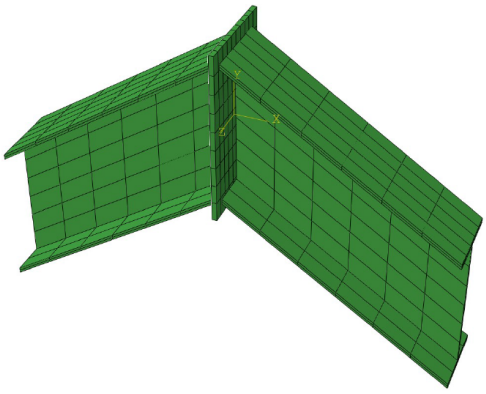


(b) Low inclination angle

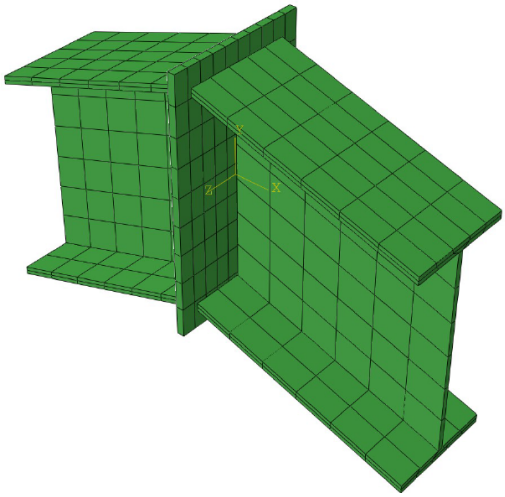


(c) Connection placed at bottom chord

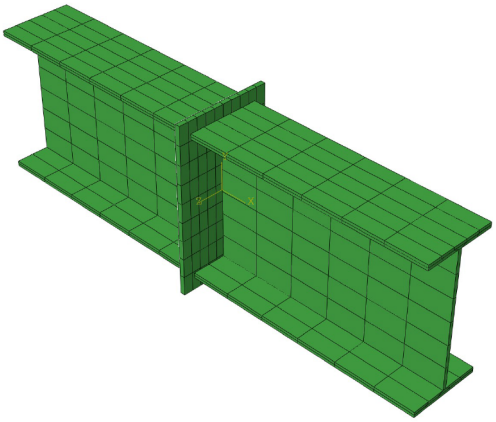
Figure 50: Different positions of the connection.



(a) Export High inclination angle



(b) Export Low inclination angle



(c) Export Connection placed at bottom chord

Figure 51: Connections exported to Abaqus

5 Discussion and Further Work

The approach, which is limited to eight-node hexahedron, is able to export numerical models to FEA successfully. The eight-node hexahedron elements is commonly used in modeling three-dimensional solids, which means the approach is able to cover a large variety of design. Its fully parametric, updating the numerical model in real time. This enables designers to quickly run simulation without having to redefine the model for each modification. During conceptional design, the engineer is able to to run several analyzes over a shorter time period.

The final product became more universal than the problem description stated, it's able to disrectize most solids and are not exclusively for joint, this creates a foundation for future development. By supplementing it with a series of predefined solutions, similar to that of the IPE component and the predefined joint, it's possible to create a library which can cover a large range of designs. In the future, it could be possible to create a modules which is related to EC3. By creating a connection between the modules of the library and a component which calculates the component combination in accordance to EC3 (fig.52).

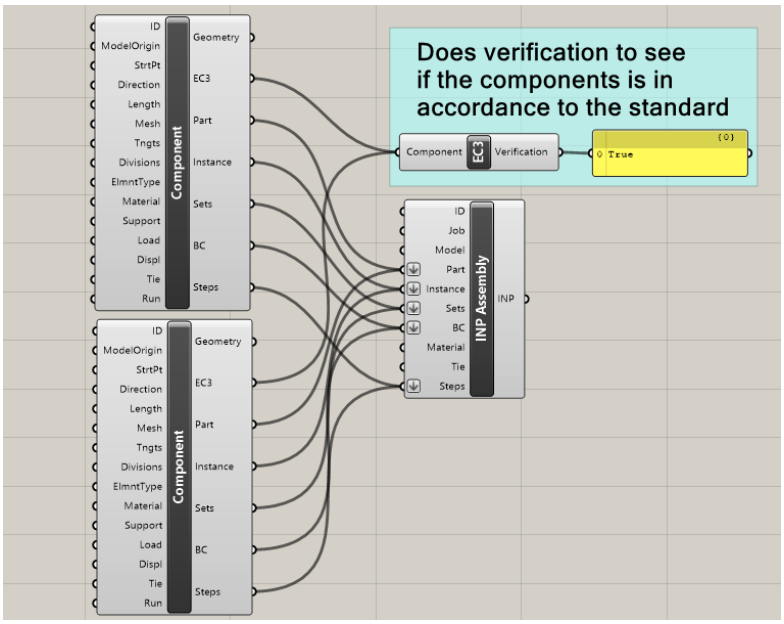


Figure 52: Possible connection to EC3

The majority of the work done on this paper was on the method on how to export the building information, and less on interpretation of simulation results. A future study on assembling stable numerical models is required. Through doing a series of simulations, it could be possible to optimize the boundary conditions of the predefined connections so that they cover most cases.

The mesh discretization is controlled by the user through the available inputs: mesh, curve divisions and tangents. This enables the user to control the local refinement of mesh giving the analyzes higher accuracy. While the use of mesh generators are faster, they might in some instances result in some poor meshes which causes poor results from the FEA.

For this approach, it would be possible to include a scaling factor list similar to the tangent list (refer section 3.5). This could be a list of factors which performs a non-uniform scaling, or uniform, on the surface elements in order to create a non-uniform cross section member. During the initial transformation process (translation and rotation), the surface elements are scaled according to a given point on the curve.

As the approach is now it's time costly to create the models for more complex joints. The components in it has to be discretized individually and then be assembled. In the case of bolted connections, the mesh surrounding the hole has to be created locally and then integrated in the global mesh of the component. Given time, this could be solved and then that solution could be made parametric and applied on future parts.

6 Conclusion

It's been verified that a computer aided three-dimensional interactive application can be used to obtain direct contact to a Finite Element Software, and that it can be applied to export building information models from CAD to a finite element analyzes.

The approach derived in this paper enables engineers to create numerical models which adapts itself according to a design, models which can be directly be exported to an finite element analyzes. Its universal and versatile, giving the possibility to cover a large variety of design problems. In the future it can be supplemented with a library of predefined designs for a more complete solution package.

References

[1] Adams, R. A., Essex, C., 2010. Calculus, A Complete Course, Seventh Edition.

[2] Aish, R., 2013. First Build Your Tools. INSIDE SMARTGEOMETRY Expanding the Architectural Possibilitiees of Computational Design.

[3] Apolinarska, A. A., Bärtschi, R., Furrer, R., Gramazio, F., Kohler, M., 2016. Mastering the "Sequential Roof". Advances in Architectural Geometry 2016.

[4] Bell, K., 2013. An engineering approach to Finite Element Analysis.

[5] Bletzinger, K.-U., Ramm, E., 2014. Computational form finding and optimization. SHELL STRUCTURES FOR ARCHITECTURE Form Finding and Optimization.

[6] Cook, R. D., Malkus, D. S., Plesha, M. E., Witt, R. J., 2002. Concepts And Applications Of Finite Element Analysis, Fourth Edition.

[7] Goldup, K., Kostura, Z., Tavolaro, T., Wolfe, S., 2017. Advanced Engineering With Building Information Modeling. Workflows: Expanding Architecture's Territory in the Design and Delivery of Buildings.

[8] Larsen, P. K., 2013. Dimensjonering av stålkonstruksjoner, 2.utg.

[9] Maddock, R., Kestelier, X. D., Smith, R. R., Haylock, D., 2017. Maggie's at the Robert Parfett Building, Manchester. Fabricate 2017: Rethinking Design and Construction.

[10] Mode Lab, 2015. The Grasshopper Primer. Third edition v3.3 Edition.

[11] Systemes Dassault, 2014. Abaqus CAE User's Guide. Abaqus 6.14 Edition.

[12] Warton, J., May, H., 2017. Automated DesignToFabrication for Architectural Envelopes a Stadium Skin Case Study. Fabricate 2017: Rethinking Design and Construction.

Appendix

A Code scripts

A.1 Extrude Part

```
//Declare variables;
    Point3d Crd;
    Vector3d Trns, Tngt, xprd;
    Plane pln;
    Transform Scle, rot;
    double Scly, Sclz;
    string Nr, X, Y, Z;
    int Count;

    //Create output tree;
    DataTree <Curve> Crvs = new DataTree<Curve>();
    DataTree <Point3d> Elements = new DataTree<Point3d>();
    DataTree <System.Object> GeoTree = new DataTree<System.Object>();
    DataTree <string> Inf = new DataTree<string>();
    DataTree <int> Tick = new DataTree<int>();
    DataTree <int> Tock = new DataTree<int>();

    //Create Lists;
    List< Point3d > Vertices = new List<Point3d>();
    List< Point3d > Nds = new List<Point3d>();
    List< Brep > Srf = new List<Brep>();
    List< Brep > ObjList = new List<Brep>();
    List< string > PrtLst = new List<string>();
    List< string > SetLst = new List<string>();
    List< string > InsLst = new List<string>();
    List< string > BCLst = new List<string>();
    List< string > StpLst = new List<string>();
    List< string > ElNr = new List<string>();
```

```
List<Brep> SupLst = new List<Brep>();
List<double> TolSupLst = new List<double>();
List<Brep> DisLst = new List<Brep>();
List<double> TolDisLst = new List<double>();
List<Brep> LdLst = new List<Brep>();
List<double> TolLdLst = new List<double>();
List<Brep> TieLst = new List<Brep>();
List<double> TolTieLst = new List<double>();
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
//Create Curve according to input;
Direction.Unitize();
Point3d EndPt = new Point3d(StrtPt.X + Length * Direction.X, StrtP
Curve Rail = new LineCurve(StrtPt, EndPt);
//Divide the curve to get points where the loft curves will be pla
var PrPts = Rail.DivideByCount(Divisions, true);

for ( int i = 0; i < Divisions + 1; i++)
{
  Trns = Rail.PointAt(PrPts[i]) - new Point3d(0, 0, 0);
  if (i < Tngts.Count && Tngts.Count > 0)
  {
    Tngt = Tngts[i];
    xprd = Vector3d.CrossProduct(Rail.TangentAt(PrPts[i]), Tngts[i
    pln = new Plane(Rail.PointAt(PrPts[i]), xprd);
    Scly = 1 / Math.Cos(Vector3d.VectorAngle(Rail.TangentAt(PrPts[
    if (xprd == new Vector3d(0, 0, 0))
    {
      Sclz = 1.0;
    }
    else
    {
      Sclz = 1 / Math.Cos(Vector3d.VectorAngle(Rail.TangentAt(PrPt
    }
  }
  else
  {
    Tngt = Rail.TangentAt(PrPts[i]);
    Scly = 1.0;
    Sclz = 1.0;
  }
```

```
    Scle = Transform.Scale(Plane.WorldYZ, Scly, Sclz, 1);
    rot = Transform.Rotation(Vector3d.XAxis, Tngt, Rail.PointAt(PrPt
    Brep Brds = Mesh.Branch(0)[0].DuplicateBrep();
    Brds.Transform(Scle);
    Brds.Translate(Trns);
    Brds.Transform(rot);
    if (i == 0 || i == Divisions)
    {
      Srf.Add(Brds);
    }
    Crvs.AddRange(Brds.DuplicateEdgeCurves(), new GH_Path(i));
    for ( int j = 0; j < Mesh.Branch(1).Count; j++)
    {
      Brep Obj = Mesh.Branch(1)[j].DuplicateBrep();
      Obj.Transform(Scle);
      Obj.Translate(Trns);
      Obj.Transform(rot);
      ObjList.Add(Obj);
      Vertices.AddRange(Obj.DuplicateVertices());
      Elements.AddRange(Obj.DuplicateVertices(), new GH_Path(0, i, j
      GeoTree.AddRange(Obj.DuplicateEdgeCurves(), new GH_Path(1, 0,
    }
  }
  //Divide the curve to get points where the loft curves will be pla
  if (Run)
  {
    for ( int i = 0; i < Divisions; i++)
    {
      for ( int j = 0; j < Crvs.Branch(i).Count; j++)
      {
        Srf.AddRange(Brep.CreateFromLoft(new List<Curve>{ Crvs.Branc
      }
    }
    Brep Shape = Brep.JoinBreps(Srf, doc.ModelAbsoluteTolerance)[0];
    GeoTree.Add(Shape, new GH_Path(0, 0, 0));
    //Discretize the data of the Part;
    PrtLst.Add(string.Format("*Part, name={0}", ID));
    //Nodes of the Part;
    PrtLst.Add("*Node");
    Nds.AddRange(Point3d.CullDuplicates(Vertices, doc.ModelAbsoluteT
    for ( int i = 0; i < Nds.Count; i++)
```

```
{
  Nr = (i + 1).ToString();
  if (ModelOrigin != Point3d.Origin)
  {
    Crd = new Point3d(Nds[i].X - ModelOrigin.X, Nds[i].Y - Model
  }
  else
  {
    Crd = Nds[i];
  }
  X = Crd.X.ToString();
  Y = Crd.Y.ToString();
  Z = Crd.Z.ToString();
  PrtLst.Add(Nr + ", " + X + ", " + Z + ", " + Y);
}
//Sort element Nodes and identify faces which is supported or loa
PrtLst.Add(string.Format("*Element, type={0}", ElmntType[0]));
Count = 1;
//If Load, import input for load;
if ( Load.PathExists(0) )
{
  double TolLd;
  Brep LdSrf = new Brep();
  for (int i = 0; i < Load.Branch(0).Count(); i++)
  {
    GH_Convert.ToBrep(Load.Branch(0)[i], ref LdSrf, Grasshopper.
    GH_Convert.ToDouble(Load.Branch(3)[i], out TolLd, GH_Convers
    TolLdLst.Add(TolLd);
    LdLst.Add(LdSrf);
  }
}
//If Tie, import input for tie;
if ( Tie.PathExists(0) )
{
  double TolTie;
  Brep TieSrf = new Brep();
  for (int i = 0; i < Tie.Branch(0).Count(); i++)
  {
    GH_Convert.ToBrep(Tie.Branch(0)[i], ref TieSrf, Grasshopper.
    GH_Convert.ToDouble(Tie.Branch(2)[i], out TolTie, GH_Convers
    TolTieLst.Add(TolTie);
```

```
      TieLst.Add(TieSrf);
    }
  }

  for ( int i = 0; i < Divisions; i++)
  {
    for ( int j = 0; j < Mesh.Branch(1).Count; j++)
    {
      ElNr.Add(Count.ToString());
      for ( int k = 0; k < Elements.Branch(0, i, j).Count; k++)
      {
        for ( int l = 0; l < Nds.Count; l++)
        {
          if ( PtCmp(Elements.Branch(0, i, j)[k], Nds[l]))
          {
            ElNr.Add((l + 1).ToString());
          }
        }
      }
      for ( int k = 0; k < Elements.Branch(0, i, j).Count; k++)
      {
        for ( int l = 0; l < Nds.Count; l++)
        {
          if ( PtCmp(Elements.Branch(0, i + 1, j)[k], Nds[l]))
          {
            ElNr.Add((l + 1).ToString());
          }
        }
        if ( Load.PathExists(0) )
        {
          for (int m = 0; m < Load.Branch(0).Count; m++)
          {
            Tick.EnsurePath(m);
            if ( IdenSet(LdLst[m], Elements.Branch(0, i, j)[k], To
            {
              Tick.Add(k + 1, new GH_Path(m));
              if (Tick.Branch(m).Count == 4)
              {
                Inf.Add((Count).ToString() + ",", new GH_Path(1, m
                Inf.Add(SrfInd(Tick.Branch(m)).ToString(), new GH_
              }
```

```
        }
        if ( IdenSet(LdLst[m], Elements.Branch(0, i + 1, j)[k]
        {
          Tick.Add(k + 5, new GH_Path(m));
          if (Tick.Branch(m).Count == 4)
          {
            Inf.Add((Count).ToString() + ",", new GH_Path(1, m
            Inf.Add(SrfInd(Tick.Branch(m)).ToString(), new GH_
          }
        }
      }
    }
    if ( Tie.PathExists(0) )
    {
      for (int m = 0; m < Tie.Branch(0).Count; m++)
      {
        Tock.EnsurePath(m);
        if ( IdenSet(TieLst[m], Elements.Branch(0, i, j)[k], T
        {
          Tock.Add(k + 1, new GH_Path(m));
          if (Tock.Branch(m).Count == 4)
          {
            Inf.Add((Count).ToString() + ",", new GH_Path(1, m
            Inf.Add(SrfInd(Tock.Branch(m)).ToString(), new GH_
          }
        }
        if ( IdenSet(TieLst[m], Elements.Branch(0, i + 1, j)[k
        {
          Tock.Add(k + 5, new GH_Path(m));
          if (Tock.Branch(m).Count == 4)
          {
            Inf.Add((Count).ToString() + ",", new GH_Path(1, m
            Inf.Add(SrfInd(Tock.Branch(m)).ToString(), new GH_
          }
        }
      }
    }
  }
}
PrtLst.Add(String.Join(",  ", ElNr.ToArray()));
Count = Count + 1;
ElNr.Clear();
```

```
    Tick.ClearData();
    Tock.ClearData();
  }
}
//Identify nodes on supported face;
if ( Support.PathExists(0) )
{
  double TolSup;
  Brep SupSrf = new Brep();
  for (int i = 0; i < Support.Branch(0).Count(); i++)
  {
    GH_Convert.ToBrep(Support.Branch(0)[i], ref SupSrf, Grasshop
    GH_Convert.ToDouble(Support.Branch(3)[i], out TolSup, GH_Co
    TolSupLst.Add(TolSup);
    SupLst.Add(SupSrf);
  }
  for (int i = 0; i < Nds.Count; i++)
  {
    for (int j = 0; j < Support.Branch(0).Count; j++)
    {
      if ( IdenSet(SupLst[j], Nds[i], TolSupLst[j]) )
      {
        Inf.Add((i + 1).ToString() + ",", new GH_Path(0, j, 1));
        GeoTree.Add(Nds[i].X.ToString(), new GH_Path(2, j, 0));
        GeoTree.Add(Nds[i].Y.ToString(), new GH_Path(2, j, 1));
        GeoTree.Add(Nds[i].Z.ToString(), new GH_Path(2, j, 2));
      }
    }
  }
}
//Identify displaced nodes;
if ( Displ.PathExists(0) )
{
  double TolDis;
  Brep DisSrf = new Brep();
  for (int i = 0; i < Displ.Branch(0).Count(); i++)
  {
    GH_Convert.ToBrep(Displ.Branch(0)[i], ref DisSrf, Grasshoppe
    GH_Convert.ToDouble(Displ.Branch(3)[i], out TolDis, GH_Conv
    TolDisLst.Add(TolDis);
    DisLst.Add(DisSrf);
```

```
    }
    for (int i = 0; i < Nds.Count; i++)
    {
      for (int j = 0; j < Displ.Branch(0).Count; j++)
      {
        if ( IdenSet(DisLst[j], Nds[i], TolDisLst[j]) )
        {
          Inf.Add((i + 1).ToString() + ",", new GH_Path(0, j, 2));
          GeoTree.Add(Nds[i].X.ToString(), new GH_Path(3, j, 0));
          GeoTree.Add(Nds[i].Y.ToString(), new GH_Path(3, j, 1));
          GeoTree.Add(Nds[i].Z.ToString(), new GH_Path(3, j, 2));
        }
      }
    }
}
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
//Create sets of nodes;
PrtLst.Add("*Nset, nset=Set-1, generate");
PrtLst.Add(string.Format("   1,    {0},    1", Nds.Count));
//Create sets of elements;
PrtLst.Add("*Elset, elset=Set-1, generate");
PrtLst.Add(string.Format("   1,    {0},    1", Count - 1));
//Define section;
PrtLst.Add(string.Format("**Section: {0}_section", ID));
PrtLst.Add(string.Format("*Solid Section, elset=Set-1, material=
PrtLst.Add(",");
//Finish defining Part;
PrtLst.Add("*End Part");
PrtLst.Add("**");
//Create instance of the part;
InsLst.Add(string.Format("*Instance, name={0}, part={0}", ID));
InsLst.Add("*End Instance");
InsLst.Add("**");
//BC; Add nodes on supported Face;
if ( Inf.PathExists(0, 0, 1) )
{
  for (int i = 0; i < Support.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Nset, nset={0}, instance={1}", Su
    for (int j = 0; j < Inf.Branch(0, i, 1).Count; j++)
```

```
    {
      SetLst.Add(Inf.Branch(0, i, 1)[j]);
    }
  }
}
//Displ/Rot; Add displaced nodes;
if ( Inf.PathExists(0, 0, 2) )
{
  for (int i = 0; i < Displ.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Nset, nset={0}, instance={1}", Di
    for (int j = 0; j < Inf.Branch(0, i, 2).Count; j++)
    {
      SetLst.Add(Inf.Branch(0, i, 2)[j]);
    }
  }
}
//Add elements on tied face;
if ( Inf.PathExists(1, 0, 4) )
{
  for (int i = 0; i < Tie.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Elset, elset={0}-{1}-TieSrf, inte
    for (int j = 0; j < Inf.Branch(1, i, 4).Count; j++)
    {
      SetLst.Add(Inf.Branch(1, i, 4)[j]);
    }
  }
}
//Define tie surface;
if ( Inf.PathExists(1, 0, 5) )
{
  for (int i = 0; i < Tie.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Surface, type=ELEMENT, name={0}-{
    SetLst.Add(string.Format("{0}-{1}-TieSrf, S{2}", Tie.Branch(
  }
}
//Add elements on loaded face;
if ( Inf.PathExists(1, 0, 2) )
{
```

```
  for (int i = 0; i < Load.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Elset, elset={0}-Surf, internal,
    for (int j = 0; j < Inf.Branch(1, i, 2).Count; j++)
    {
      SetLst.Add(Inf.Branch(1, i, 2)[j]);
    }
  }
}
//Define load surface;
if ( Inf.PathExists(1, 0, 2) )
{
  for (int i = 0; i < Load.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Surface, type=ELEMENT, name={0}",
    SetLst.Add(string.Format("{0}-Surf, S{1}", Load.Branch(2)[i]
  }
}
//Boundary conditions;
if ( Support.PathExists(0) )
{
  for (int i = 0; i < Support.Branch(1).Count; i++)
  {
    BCLst.Add(string.Format("{0}", Support.Branch(1)[i]));
  }
}
//Steps;
if ( Load.PathExists(0) )
{
  for (int i = 0; i < Load.Branch(1).Count; i++)
  {
    StpLst.Add(string.Format("{0}", Load.Branch(1)[i]));;
  }
}
if ( Displ.PathExists(0) )
{
  for (int i = 0; i < Displ.Branch(1).Count; i++)
  {
    StpLst.Add(string.Format("{0}", Displ.Branch(1)[i]));
  }
}
```

```
  }
  Geometry = GeoTree;
  Part = PrtLst;
  Instance = InsLst;
  Sets = SetLst;
  BC = BCLst;
  Steps = StpLst;
}

// <Custom additional code>

public int SrfInd(List < int > N)
{
  int Indx;

  if (!N.Except(new List<int>{ 1, 2, 3, 4 }).Any() && N.Count == 4)
  {
    Indx = 1;
  }
  else if(!N.Except(new List<int>{ 5, 6, 7, 8 }).Any() && N.Count ==
  {
    Indx = 2;
  }
  else if(!N.Except(new List<int>{ 1, 2, 5, 6 }).Any() && N.Count ==
  {
    Indx = 3;
  }
  else if(!N.Except(new List<int>{ 2, 3, 6, 7 }).Any() && N.Count ==
  {
    Indx = 4;
  }
  else if(!N.Except(new List<int>{ 3, 4, 7, 8 }).Any() && N.Count ==
  {
    Indx = 5;
  }
  else
  {
    Indx = 6;
  }
  return Indx;
```

```
}

public bool IdenSet(Brep Srf, Point3d Nd, double Tol)
{
  bool Truth;
  Point3d Pt = Srf.ClosestPoint(Nd);
  Vector3d Vct = Pt - Nd;

  if (Vct.Length < Tol)
  {
    Truth = true;
  }
  else
  {
    Truth = false;
  }
  return Truth;
}

public bool PtCmp(Point3d ElNd, Point3d Nd)
{
  bool Truth;
  Vector3d Vct = ElNd - Nd;

  if (Vct.Length < doc.ModelAbsoluteTolerance)
  {
    Truth = true;
  }
  else
  {
    Truth = false;
  }
  return Truth;
}
```

A.2 Sweep Part

```
//Declare variables;
    Point3d Crd;
    Vector3d Trns, Tngt, xprd;
    Plane pln;
    Transform Scle, rot;
    double Scly, Sclz;
    string Nr, X, Y, Z;
    int Count;

    //Create output tree;
    DataTree <Curve> Crvs = new DataTree<Curve>();
    DataTree <Point3d> Elements = new DataTree<Point3d>();
    DataTree <System.Object> GeoTree = new DataTree<System.Object>();
    DataTree <string> Inf = new DataTree<string>();
    DataTree <int> Tick = new DataTree<int>();
    DataTree <int> Tock = new DataTree<int>();

    //Create Lists;
    List< Point3d > Vertices = new List<Point3d>();
    List< Point3d > Nds = new List<Point3d>();
    List< Brep > Srf = new List<Brep>();
    List< Brep > ObjList = new List<Brep>();
    List< string > PrtLst = new List<string>();
    List< string > SetLst = new List<string>();
    List< string > InsLst = new List<string>();
    List< string > BCLst = new List<string>();
    List< string > StpLst = new List<string>();
    List< string > ElNr = new List<string>();
    List<Brep> SupLst = new List<Brep>();
    List<double> TolSupLst = new List<double>();
    List<Brep> DisLst = new List<Brep>();
    List<double> TolDisLst = new List<double>();
    List<Brep> LdLst = new List<Brep>();
    List<double> TolLdLst = new List<double>();
    List<Brep> TieLst = new List<Brep>();
    List<double> TolTieLst = new List<double>();
    ////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////
    //Divide the curve to get points where the loft curves will be pla
```

```
var PrPts = Rail.DivideByCount(Divisions, true);

for (int i = 0; i < Divisions + 1; i++)
{
  Trns = Rail.PointAt(PrPts[i]) - new Point3d(0, 0, 0);
  if (i < Tngts.Count && Tngts.Count > 0)
  {
    Tngt = Tngts[i];
    xprd = Vector3d.CrossProduct(Rail.TangentAt(PrPts[i]), Tngts[i
    pln = new Plane(Rail.PointAt(PrPts[i]), xprd);
    Scly = 1 / Math.Cos(Vector3d.VectorAngle(Rail.TangentAt(PrPts[
    if (xprd == new Vector3d(0, 0, 0))
    {
      Sclz = 1.0;
    }
    else
    {
      Sclz = 1 / Math.Cos(Vector3d.VectorAngle(Rail.TangentAt(PrPt
    }
  }
  else
  {
    Tngt = Rail.TangentAt(PrPts[i]);
    Scly = 1.0;
    Sclz = 1.0;
  }
  Scle = Transform.Scale(Plane.WorldYZ, Scly, Sclz, 1);
  rot = Transform.Rotation(Vector3d.XAxis, Tngt, Rail.PointAt(PrPt
  Brep Brds = Mesh.Branch(0)[0].DuplicateBrep();
  Brds.Transform(Scle);
  Brds.Translate(Trns);
  Brds.Transform(rot);
  if (i == 0 || i == Divisions)
  {
    Srf.Add(Brds);
  }
  Crvs.AddRange(Brds.DuplicateEdgeCurves(), new GH_Path(i));
  for (int j = 0; j < Mesh.Branch(1).Count; j++)
  {
    Brep Obj = Mesh.Branch(1)[j].DuplicateBrep();
    Obj.Transform(Scle);
```

```
      Obj.Translate(Trns);
      Obj.Transform(rot);
      ObjList.Add(Obj);
      Vertices.AddRange(Obj.DuplicateVertices());
      Elements.AddRange(Obj.DuplicateVertices(), new GH_Path(0, i, j
      GeoTree.AddRange(Obj.DuplicateEdgeCurves(), new GH_Path(1, 0,
    }
  }
  //Divide the curve to get points where the loft curves will be pla
  if (Run)
  {
    for ( int i = 0; i < Divisions; i++)
    {
      for ( int j = 0; j < Crvs.Branch(i).Count; j++)
      {
        Srf.AddRange(Brep.CreateFromLoft(new List<Curve>{ Crvs.Branc
      }
    }
    Brep Shape = Brep.JoinBreps(Srf, doc.ModelAbsoluteTolerance)[0];
    GeoTree.Add(Shape, new GH_Path(0, 0, 0));
    //Discretize the data of the Part;
    PrtLst.Add(string.Format("*Part, name={0}", ID));
    //Nodes of the Part;
    PrtLst.Add("*Node");
    Nds.AddRange(Point3d.CullDuplicates(Vertices, doc.ModelAbsoluteT
    for ( int i = 0; i < Nds.Count; i++)
    {
      Nr = (i + 1).ToString();
      if (ModelOrigin != Point3d.Origin)
      {
        Crd = new Point3d(Nds[i].X - ModelOrigin.X, Nds[i].Y - Model
      }
      else
      {
        Crd = Nds[i];
      }
      X = Crd.X.ToString();
      Y = Crd.Y.ToString();
      Z = Crd.Z.ToString();
      PrtLst.Add(Nr + ", " + X + ", " + Z + ", " + Y);
    }
```

```
//Sort element Nodes and identify faces which is supported or loa
PrtLst.Add(string.Format("*Element, type={0}", ElmntType[0]));
Count = 1;
//If Load, import input for load;
if ( Load.PathExists(0) )
{
  double TolLd;
  Brep LdSrf = new Brep();
  for (int i = 0; i < Load.Branch(0).Count(); i++)
  {
    GH_Convert.ToBrep(Load.Branch(0)[i], ref LdSrf, Grasshopper.
    GH_Convert.ToDouble(Load.Branch(3)[i], out TolLd, GH_Convers
    TolLdLst.Add(TolLd);
    LdLst.Add(LdSrf);
  }
}
//If Tie, import input for tie;
if ( Tie.PathExists(0) )
{
  double TolTie;
  Brep TieSrf = new Brep();
  for (int i = 0; i < Tie.Branch(0).Count(); i++)
  {
    GH_Convert.ToBrep(Tie.Branch(0)[i], ref TieSrf, Grasshopper.
    GH_Convert.ToDouble(Tie.Branch(2)[i], out TolTie, GH_Convers
    TolTieLst.Add(TolTie);
    TieLst.Add(TieSrf);
  }
}

for ( int i = 0; i < Divisions; i++)
{
  for ( int j = 0; j < Mesh.Branch(1).Count; j++)
  {
    ElNr.Add(Count.ToString());
    for ( int k = 0; k < Elements.Branch(0, i, j).Count; k++)
    {
      for ( int l = 0; l < Nds.Count; l++)
      {
        if ( PtCmp(Elements.Branch(0, i, j)[k], Nds[l]))
        {
```

```
        ElNr.Add((l + 1).ToString());
      }
    }
  }
  for ( int k = 0; k < Elements.Branch(0, i, j).Count; k++)
  {
    for ( int l = 0; l < Nds.Count; l++)
    {
      if ( PtCmp(Elements.Branch(0, i + 1, j)[k], Nds[l]))
      {
        ElNr.Add((l + 1).ToString());
      }
    }
    if ( Load.PathExists(0) )
    {
      for (int m = 0; m < Load.Branch(0).Count; m++)
      {
        Tick.EnsurePath(m);
        if ( IdenSet(LdLst[m], Elements.Branch(0, i, j)[k], To
        {
          Tick.Add(k + 1, new GH_Path(m));
          if (Tick.Branch(m).Count == 4)
          {
            Inf.Add((Count).ToString() + ",", new GH_Path(1, m
            Inf.Add(SrfInd(Tick.Branch(m)).ToString(), new GH_
          }
        }
        if ( IdenSet(LdLst[m], Elements.Branch(0, i + 1, j)[k]
        {
          Tick.Add(k + 5, new GH_Path(m));
          if (Tick.Branch(m).Count == 4)
          {
            Inf.Add((Count).ToString() + ",", new GH_Path(1, m
            Inf.Add(SrfInd(Tick.Branch(m)).ToString(), new GH_
          }
        }
      }
    }
    if ( Tie.PathExists(0) )
    {
      for (int m = 0; m < Tie.Branch(0).Count; m++)
```

```
            {
              Tock.EnsurePath(m);
              if ( IdenSet(TieLst[m], Elements.Branch(0, i, j)[k], T
              {
                Tock.Add(k + 1, new GH_Path(m));
                if (Tock.Branch(m).Count == 4)
                {
                  Inf.Add((Count).ToString() + ",", new GH_Path(1, m
                  Inf.Add(SrfInd(Tock.Branch(m)).ToString(), new GH_
                }
              }
              if ( IdenSet(TieLst[m], Elements.Branch(0, i + 1, j)[k
              {
                Tock.Add(k + 5, new GH_Path(m));
                if (Tock.Branch(m).Count == 4)
                {
                  Inf.Add((Count).ToString() + ",", new GH_Path(1, m
                  Inf.Add(SrfInd(Tock.Branch(m)).ToString(), new GH_
                }
              }
            }
          }
        }
        PrtLst.Add(String.Join(",  ", ElNr.ToArray()));
        Count = Count + 1;
        ElNr.Clear();
        Tick.ClearData();
        Tock.ClearData();
      }
    }
    //Identify nodes on supported face;
    if ( Support.PathExists(0) )
    {
      double TolSup;
      Brep SupSrf = new Brep();
      for (int i = 0; i < Support.Branch(0).Count(); i++)
      {
        GH_Convert.ToBrep(Support.Branch(0)[i], ref SupSrf, Grasshop
        GH_Convert.ToDouble(Support.Branch(3)[i], out TolSup, GH_Co
        TolSupLst.Add(TolSup);
        SupLst.Add(SupSrf);
```

```
    }
    for (int i = 0; i < Nds.Count; i++)
    {
      for (int j = 0; j < Support.Branch(0).Count; j++)
      {
        if ( IdenSet(SupLst[j], Nds[i], TolSupLst[j]) )
        {
          Inf.Add((i + 1).ToString() + ",", new GH_Path(0, j, 1));
          GeoTree.Add(Nds[i].X.ToString(), new GH_Path(2, j, 0));
          GeoTree.Add(Nds[i].Y.ToString(), new GH_Path(2, j, 1));
          GeoTree.Add(Nds[i].Z.ToString(), new GH_Path(2, j, 2));
        }
      }
    }
  }
  //Identify displaced nodes;
  if ( Displ.PathExists(0) )
  {
    double TolDis;
    Brep DisSrf = new Brep();
    for (int i = 0; i < Displ.Branch(0).Count(); i++)
    {
      GH_Convert.ToBrep(Displ.Branch(0)[i], ref DisSrf, Grasshoppe
      GH_Convert.ToDouble(Displ.Branch(3)[i], out TolDis, GH_Conv
      TolDisLst.Add(TolDis);
      DisLst.Add(DisSrf);
    }
    for (int i = 0; i < Nds.Count; i++)
    {
      for (int j = 0; j < Displ.Branch(0).Count; j++)
      {
        if ( IdenSet(DisLst[j], Nds[i], TolDisLst[j]) )
        {
          Inf.Add((i + 1).ToString() + ",", new GH_Path(0, j, 2));
          GeoTree.Add(Nds[i].X.ToString(), new GH_Path(3, j, 0));
          GeoTree.Add(Nds[i].Y.ToString(), new GH_Path(3, j, 1));
          GeoTree.Add(Nds[i].Z.ToString(), new GH_Path(3, j, 2));
        }
      }
    }
  }
```

```
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
//Create sets of nodes;
PrtLst.Add("*Nset, nset=Set-1, generate");
PrtLst.Add(string.Format("    1,     {0},     1", Nds.Count));
//Create sets of elements;
PrtLst.Add("*Elset, elset=Set-1, generate");
PrtLst.Add(string.Format("    1,     {0},     1", Count - 1));
//Define section;
PrtLst.Add(string.Format("**Section: {0}_section", ID));
PrtLst.Add(string.Format("*Solid Section, elset=Set-1, material=
PrtLst.Add(",");
//Finish defining Part;
PrtLst.Add("*End Part");
PrtLst.Add("**");
//Create instance of the part;
InsLst.Add(string.Format("*Instance, name={0}, part={0}", ID));
InsLst.Add("*End Instance");
InsLst.Add("**");
//BC; Add nodes on supported Face;
if ( Inf.PathExists(0, 0, 1) )
{
  for (int i = 0; i < Support.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Nset, nset={0}, instance={1}", Su
    for (int j = 0; j < Inf.Branch(0, i, 1).Count; j++)
    {
      SetLst.Add(Inf.Branch(0, i, 1)[j]);
    }
  }
}
//Displ/Rot; Add displaced nodes;
if ( Inf.PathExists(0, 0, 2) )
{
  for (int i = 0; i < Displ.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Nset, nset={0}, instance={1}", Di
    for (int j = 0; j < Inf.Branch(0, i, 2).Count; j++)
    {
      SetLst.Add(Inf.Branch(0, i, 2)[j]);
    }
```

```
  }
}
//Add elements on tied face;
if ( Inf.PathExists(1, 0, 4) )
{
  for (int i = 0; i < Tie.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Elset, elset={0}-{1}-TieSrf, inte
    for (int j = 0; j < Inf.Branch(1, i, 4).Count; j++)
    {
      SetLst.Add(Inf.Branch(1, i, 4)[j]);
    }
  }
}
//Define tie surface;
if ( Inf.PathExists(1, 0, 5) )
{
  for (int i = 0; i < Tie.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Surface, type=ELEMENT, name={0}-{
    SetLst.Add(string.Format("{0}-{1}-TieSrf, S{2}", Tie.Branch(
  }
}
//Add elements on loaded face;
if ( Inf.PathExists(1, 0, 2) )
{
  for (int i = 0; i < Load.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Elset, elset={0}-Surf, internal,
    for (int j = 0; j < Inf.Branch(1, i, 2).Count; j++)
    {
      SetLst.Add(Inf.Branch(1, i, 2)[j]);
    }
  }
}
//Define load surface;
if ( Inf.PathExists(1, 0, 2) )
{
  for (int i = 0; i < Load.Branch(0).Count; i++)
  {
    SetLst.Add(string.Format("*Surface, type=ELEMENT, name={0}",
```

```
            SetLst.Add(string.Format("{0}-Surf, S{1}", Load.Branch(2)[i]
          }
        }
        //Boundary conditions;
        if ( Support.PathExists(0) )
        {
          for (int i = 0; i < Support.Branch(1).Count; i++)
          {
            BCLst.Add(string.Format("{0}", Support.Branch(1)[i]));
          }
        }
        //Steps;
        if ( Load.PathExists(0) )
        {
          for (int i = 0; i < Load.Branch(1).Count; i++)
          {
            StpLst.Add(string.Format("{0}", Load.Branch(1)[i]));;
          }
        }
        if ( Displ.PathExists(0) )
        {
          for (int i = 0; i < Displ.Branch(1).Count; i++)
          {
            StpLst.Add(string.Format("{0}", Displ.Branch(1)[i]));
          }
        }

      }
      Geometry = GeoTree;
      Part = PrtLst;
      Instance = InsLst;
      Sets = SetLst;
      BC = BCLst;
      Steps = StpLst;
    }

    // <Custom additional code>

    public int SrfInd(List < int > N)
    {
      int Indx;
```

```
  if (!N.Except(new List<int>{ 1, 2, 3, 4 }).Any() && N.Count == 4)
  {
    Indx = 1;
  }
  else if(!N.Except(new List<int>{ 5, 6, 7, 8 }).Any() && N.Count ==
  {
    Indx = 2;
  }
  else if(!N.Except(new List<int>{ 1, 2, 5, 6 }).Any() && N.Count ==
  {
    Indx = 3;
  }
  else if(!N.Except(new List<int>{ 2, 3, 6, 7 }).Any() && N.Count ==
  {
    Indx = 4;
  }
  else if(!N.Except(new List<int>{ 3, 4, 7, 8 }).Any() && N.Count ==
  {
    Indx = 5;
  }
  else
  {
    Indx = 6;
  }
  return Indx;
}

public bool IdenSet(Brep Srf, Point3d Nd, double Tol)
{
  bool Truth;
  Point3d Pt = Srf.ClosestPoint(Nd);
  Vector3d Vct = Pt - Nd;

  if (Vct.Length < Tol)
  {
    Truth = true;
  }
  else
  {
    Truth = false;
```

```
  }
  return Truth;
}

public bool PtCmp(Point3d ElNd, Point3d Nd)
{
  bool Truth;
  Vector3d Vct = ElNd - Nd;

  if (Vct.Length < doc.ModelAbsoluteTolerance)
  {
    Truth = true;
  }
  else
  {
    Truth = false;
  }
  return Truth;
}
```

A.3 Mesh Hex8

```
//Declare variables
    Vector3d Norm, y0, z0, Trns;
    Transform xform;
    double ycrd, zcrd, rad;
    Brep Obj;

    //Create output tree
    DataTree <System.Object> tree = new DataTree<System.Object>();

    //Create Lists
    List< Point3d > Vrt = new List<Point3d>();
    List< Point3d > Pts = new List<Point3d>();
    List< double > Ang = new List<double>();
    ////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////
    //Orient Mesh and Profile according to the YZ-system
    Norm = (Section.Surfaces[0]).NormalAt(0.5, 0.5);

    if(Norm == Vector3d.ZAxis)
    {
      y0 = Vector3d.CrossProduct(Vector3d.YAxis, Norm);
    }
    else
    {
      y0 = Vector3d.CrossProduct(Vector3d.ZAxis, Norm);
    }
    z0 = Norm;
    z0.Rotate(1.5 * Math.PI, y0);
    xform = Transform.Rotation(Norm,     z0, y0, Vector3d.XAxis, Vector
    Section.Transform(xform);
    Point3d Cntr = AreaMassProperties.Compute(Section).Centroid;

    Point3d pt = new Point3d(0, Offset1, Offset2);
    Trns = pt - Cntr;

    Section.Translate(Trns);
    tree.Add(Section, new GH_Path(0));

    for( int i = 0; i < IniMesh.Count(); i++)
```

```
 {
   IniMesh[i].Transform(xform);
   IniMesh[i].Translate(Trns);
 }
 ////////////////////////////////////////////////////////////////////
 ////////////////////////////////////////////////////////////////////
 //Sort vertices according to the yz-coordinate
//system w/points in the 3rd quadrant being first pt
 for( int i = 0; i < IniMesh.Count(); i++)
 {
   Cntr = AreaMassProperties.Compute(IniMesh[i]).Centroid;
   for( int j = 0; j < IniMesh[i].DuplicateVertices().Count(); j++)
   {
     Vrt.Add(IniMesh[i].DuplicateVertices()[j]);
     ycrd = Vrt[j].Y - Cntr.Y;
     zcrd = Vrt[j].Z - Cntr.Z;
     rad = Math.Atan2(zcrd, ycrd);
     Ang.Add(rad);
   }
   var Sorted = Vrt.Zip(Ang, (x, y) => new { x, y })
     .OrderBy(pair => pair.y)
     .Reverse()
     .ToList();
   Pts = Sorted.Select(pair => pair.x).ToList();
   Obj = Brep.CreateFromCornerPoints(Pts[0], Pts[1], Pts[2], Pts[3]
   tree.Add(Obj, new GH_Path(1));
   Pts.Clear();
   Ang.Clear();
   Vrt.Clear();
 }

 Mesh = tree;
```

A.4 Support

```
//Create output tree
    DataTree <System.Object> tree = new DataTree<System.Object>();
    ////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////
    //Add surface and ID to be used in Part component
    tree.Add(Face, new GH_Path(0));
    tree.Add(ID, new GH_Path(2));
    ////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////
    //Add conditions to be exported in INP
    for( int i = 0; i < Conditions.Count(); i++)
    {
      if(Conditions[i] == "U1" || Conditions[i] == "1")
      {
        tree.Add(string.Format("{0}, 1, 1", ID), new GH_Path(1));
      }
      if(Conditions[i] == "U2" || Conditions[i] == "2")
      {
        tree.Add(string.Format("{0}, 2, 2", ID), new GH_Path(1));
      }
      if(Conditions[i] == "U3" || Conditions[i] == "3")
      {
        tree.Add(string.Format("{0}, 3, 3", ID), new GH_Path(1));
      }
      if(Conditions[i] == "UR1" || Conditions[i] == "4")
      {
        tree.Add(string.Format("{0}, 4, 4", ID), new GH_Path(1));
      }
      if(Conditions[i] == "UR2" || Conditions[i] == "5")
      {
        tree.Add(string.Format("{0}, 5, 5", ID), new GH_Path(1));
      }
      if(Conditions[i] == "UR3" || Conditions[i] == "6")
      {
        tree.Add(string.Format("{0}, 6, 6", ID), new GH_Path(1));
      }

    }
    ////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////
//Add tolerance for pointcompare
if(Tol == 0)
{
  tree.Add(0.001, new GH_Path(3));
}
else
{
  tree.Add(Tol, new GH_Path(3));
}

Support = tree;
```

A.5 Load

```
//Create output tree
    DataTree <System.Object> tree = new DataTree<System.Object>();
    ////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////
    //Add surface and ID to be used in Part component
    tree.Add(Face, new GH_Path(0));
    tree.Add(ID, new GH_Path(2));
    ////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////
    //Add conditions to be exported in INP
    tree.Add("**", new GH_Path(1));
    tree.Add(string.Format("** STEP: {0}", ID), new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add(string.Format("*Step, name={0}, nlgeom=NO", ID), new GH_P
    tree.Add("Uniform Load", new GH_Path(1));
    tree.Add("*Static", new GH_Path(1));
    tree.Add("0.1, 1., 1e-05, 1.", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add("** LOADS", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add(string.Format("** Name: {0}-Load    Type: Pressure", ID), 
    tree.Add("*Dsload", new GH_Path(1));
    tree.Add(string.Format("{0}, P, {1}", ID, Magnitude), new GH_Path(
    tree.Add("**", new GH_Path(1));
    tree.Add("** OUTPUT REQUESTS", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add("*Restart, write, frequency=0", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add("** FIELD OUTPUT: F-Output-1", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add("*Output, field, variable=PRESELECT", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add("** HISTORY OUTPUT: H-Output-1", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add("*Output, history, variable=PRESELECT", new GH_Path(1));
    tree.Add("*End Step", new GH_Path(1));
    ////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////
    //Add tolerance for pointcompare
```

```
if(Tol == 0)
{
  tree.Add(0.001, new GH_Path(3));
}
else
{
  tree.Add(Tol, new GH_Path(3));
}
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
Load = tree;
```

A.6 Displacement

```
//Create output tree
    DataTree <System.Object> tree = new DataTree<System.Object>();
    //Add surface and ID to be used in Part component
    tree.Add(Face, new GH_Path(0));
    tree.Add(ID, new GH_Path(2));
    //Add tolerance for pointcompare
    if(Tol == 0)
    {
      tree.Add(0.001, new GH_Path(3));
    }
    else
    {
      tree.Add(Tol, new GH_Path(3));
    }
    //Add conditions to be exported in INP
    tree.Add("**", new GH_Path(1));
    tree.Add(string.Format("** STEP: {0}", ID), new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add(string.Format("*Step, name={0}, nlgeom=NO", ID), new GH_P
    tree.Add("*Static", new GH_Path(1));
    tree.Add("0.1, 1., 1e-05, 1.", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add("** BOUNDARY CONDITIONS", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    tree.Add(string.Format("** Name: {0} Type: Displacement/Rotation",
    tree.Add("*Boundary", new GH_Path(1));
    tree.Add("**", new GH_Path(1));
    if (Tx != 0)
    {
      tree.Add(string.Format("{0}, 1, 1, {1}", ID, Tx), new GH_Path(1)
    }
    if (Tz != 0)
    {
      tree.Add(string.Format("{0}, 2, 2, {1}", ID, Tz), new GH_Path(1)
    }
    if (Ty != 0)
    {
      tree.Add(string.Format("{0}, 3, 3, {1}", ID, Ty), new GH_Path(1)
    }
```

```
if (Rx != 0)
{
  tree.Add(string.Format("{0}, 4, 4, {1}", ID, Rx), new GH_Path(1)
}
if (Rz != 0)
{
  tree.Add(string.Format("{0}, 5, 5, {1}", ID, Rz), new GH_Path(1)
}
if (Ry != 0)
{
  tree.Add(string.Format("{0}, 6, 6, {1}", ID, Ry), new GH_Path(1)
}
tree.Add("**", new GH_Path(1));
tree.Add("** OUTPUT REQUESTS", new GH_Path(1));
tree.Add("**", new GH_Path(1));
tree.Add("*Restart, write, frequency=0", new GH_Path(1));
tree.Add("**", new GH_Path(1));
tree.Add("** FIELD OUTPUT: F-Output-1", new GH_Path(1));
tree.Add("**", new GH_Path(1));
tree.Add("*Output, field, variable=PRESELECT", new GH_Path(1));
tree.Add("**", new GH_Path(1));
tree.Add("** HISTORY OUTPUT: H-Output-1", new GH_Path(1));
tree.Add("**", new GH_Path(1));
tree.Add("*Output, history, variable=PRESELECT", new GH_Path(1));
tree.Add("*End Step", new GH_Path(1));
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
Displ = tree;
```

A.7 Tie

```
//Create output tree
    DataTree <System.Object> tree = new DataTree<System.Object>();
    List< string > INPLst = new List<string>();
    //Add surface and ID to be used in Part component
    tree.Add(Face, new GH_Path(0));
    tree.Add(ID, new GH_Path(1));
    //Add tolerance for pointcompare
    if(Tol == 0)
    {
      tree.Add(0.001, new GH_Path(2));
    }
    else
    {
      tree.Add(Tol, new GH_Path(2));
    }
    //Add conditions to be exported in INP
    INPLst.Add(string.Format("** Constraint: {0}", ID));
    INPLst.Add(string.Format("*Tie, name={0}, adjust=yes, type=SURFACE
    INPLst.Add(string.Format("{0}-{1}-TieSrf, {0}-{2}-TieSrf", ID, Sla
    ////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////
    Part = tree;
    INP = INPLst;
```

A.8 Tangents

```
//Create output;
List< Vector3d > TngtLst = new List<Vector3d>();
//Create Variables;
double T;
Vector3d Vavg;
//Divide the curve to get the points;
var PrPts = Rail.DivideByCount(Divisions, true);
Tngt.Unitize();
if (Tol > 0)
{
  T = Tol;
}
else
{
  T = 0.1;
}
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
if (EndPt)
{
  TngtLst.Add(Tngt);
  Vavg = Vmn(Tngt, Rail.TangentAt(PrPts[Divisions - 1]), Rail.Tang
  TngtLst.Add(Vavg);
  for (int i = 2; i < Divisions + 1; i++)
  {
    if (1 - Math.Abs(Vector3d.Multiply(Vavg, Rail.TangentAt(PrPts[
    {
      Vavg = Rail.TangentAt(PrPts[Divisions - i]);
      TngtLst.Add(Vavg);
    }
    else
    {
      Vavg = Vmn(Vavg, Rail.TangentAt(PrPts[Divisions - i]), Rail.
      TngtLst.Add(Vavg);
    }
  }
  TngtLst.Reverse();
}
else
```

```
{
  TngtLst.Add(Tngt);
  Vavg = Vmn(Tngt, Rail.TangentAt(PrPts[1]), Rail.TangentAt(PrPts[
  TngtLst.Add(Vavg);
  for (int i = 2; i < Divisions + 1; i++)
  {
    if (1 - Math.Abs(Vector3d.Multiply(Vavg, Rail.TangentAt(PrPts[
    {
      Vavg = Rail.TangentAt(PrPts[i]);
      TngtLst.Add(Vavg);
    }
    else
    {
      Vavg = Vmn(Vavg, Rail.TangentAt(PrPts[i]), Rail.TangentAt(Pr
      TngtLst.Add(Vavg);
    }
  }
}
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
TList = TngtLst;
```

A.9 IPE

```
import rhinoscriptsyntax as rs
import Rhino as rh
##########################################################################
##########################################################################
#Create lists:
UnSclMesh = []
Mesh = []
#Create dimensions according to chosen IPE-section [NS-EN 10 034]
if IPE:
    if IPE == 'IPE80' or IPE == 'ipe80':
        d = [80, 46, 3.8, 5.2]
    elif IPE == 'IPE100' or IPE == 'ipe100':
        d = [100, 55, 4.1, 5.7]
    elif IPE == 'IPE120' or IPE == 'ipe120':
        d = [120, 64, 4.4, 6.3]
    elif IPE == 'IPE140' or IPE == 'ipe140':
        d = [140, 73, 4.7, 6.9]
    elif IPE == 'IPE160' or IPE == 'ipe160':
        d = [160, 82, 5.0, 7.4]
    elif IPE == 'IPE180' or IPE == 'ipe180':
        d = [180, 91, 5.3, 8.0]
    elif IPE == 'IPE200' or IPE == 'ipe200':
        d = [200, 100, 5.6, 8.5]
    elif IPE == 'IPE220' or IPE == 'ipe220':
        d = [220, 110, 5.9, 9.2]
    elif IPE == 'IPE240' or IPE == 'ipe240':
        d = [240, 120, 6.2, 9.8]
    elif IPE == 'IPE270' or IPE == 'ipe270':
        d = [270, 135, 6.6, 10.2]
    elif IPE == 'IPE300' or IPE == 'ipe300':
        d = [300, 150, 7.1, 10.7]
    elif IPE == 'IPE330' or IPE == 'ipe330':
        d = [330, 160, 7.5, 11.5]
    elif IPE == 'IPE360' or IPE == 'ipe360':
        d = [360, 170, 8.0, 12.7]
    elif IPE == 'IPE400' or IPE == 'ipe400':
        d = [400, 180, 8.6, 13.5]
    elif IPE == 'IPE450' or IPE == 'ipe450':
        d = [450, 190, 9.4, 14.6]
```

```
    elif IPE == 'IPE500' or IPE == 'ipe500':
        d = [500, 200, 10.2, 16.0]
    elif IPE == 'IPE550' or IPE == 'ipe550':
        d = [550, 210, 11.1, 17.2]
    elif IPE == 'IPE600' or IPE == 'ipe600':
        d = [600, 220, 12.0, 19.0]
#Draw the edges of the section
    pt = [rh.Geometry.Point3d(0,0.5*d[2],-0.5*d[0]+d[3]),
rh.Geometry.Point3d(0,0.5*d[1],-0.5*d[0]+d[3]),
rh.Geometry.Point3d(0,0.5*d[1],-0.5*d[0]+0.5*d[3]),
rh.Geometry.Point3d(0,0.5*d[1],-0.5*d[0]),
rh.Geometry.Point3d(0,-0.5*d[1],-0.5*d[0]),
rh.Geometry.Point3d(0,-0.5*d[1],-0.5*d[0]+0.5*d[3]),
rh.Geometry.Point3d(0,-0.5*d[1],-0.5*d[0]+d[3]),
rh.Geometry.Point3d(0,-0.5*d[2],-0.5*d[0]+d[3])]
    Mrr = rh.Geometry.Transform.Mirror(rh.Geometry.Plane.WorldXY)
    for i in range(0,len(pt)):
        Temppt = rh.Geometry.Point3d(pt[i].X,pt[i].Y,pt[i].Z)
        Temppt.Transform(Mrr)
        pt.append(Temppt)
    pt[8],pt[15] = pt[15],pt[8]
    pt[9],pt[14] = pt[14],pt[9]
    pt[10],pt[13] = pt[13],pt[10]
    pt[11],pt[12] = pt[12],pt[11]
    pt.append(pt[0])
    Ln = rs.AddPolyline(pt)
#Create and sort the points being used in the mesh:
    if iter and iter != 0:
        LnBot = rs.AddLine(pt[2],pt[5])
        ptsBot = rs.DivideCurve(LnBot,2*iter+1,True,True)
        LnBot2 = rs.AddLine(pt[3],pt[4])
        ptsBot2 = rs.DivideCurve(LnBot2,2*iter+1,True,True)
        for i in range(0,2*iter+1):
            Elm = rs.coercebrep(rs.AddSrfPt([ptsBot[i],ptsBot2[i],ptsB
            UnSclMesh.append(Elm)
            Elm = Elm.Duplicate()
            Elm.Transform(Mrr)
            UnSclMesh.append(Elm)
        LnFT1 = rs.AddLine(pt[1],pt[0])
        ptsFT1 = rs.DivideCurve(LnFT1,iter,True,True)
        LnFT2 = rs.AddLine(pt[6],pt[7])
```

```
            ptsFT2 = rs.DivideCurve(LnFT2,iter,True,True)
            LnW1 = rs.AddLine(pt[15],pt[0])
            ptsW1 = rs.DivideCurve(LnW1,2*iter+2,True,True)
            LnW2 = rs.AddLine(pt[8],pt[7])
            ptsW2 = rs.DivideCurve(LnW2,2*iter+2,True,True)
            Elm = rs.coercebrep(rs.AddSrfPt([ptsFT2[-1],ptsFT1[-1],ptsBot[
            UnSclMesh.append(Elm)
            Elm = Elm.Duplicate()
            Elm.Transform(Mrr)
            UnSclMesh.append(Elm)
            Elm = rs.coercebrep(rs.AddSrfPt([ptsW2[iter+1],ptsW1[iter+1],p
            UnSclMesh.append(Elm)
            Elm = Elm.Duplicate()
            Elm.Transform(Mrr)
            UnSclMesh.append(Elm)
            for i in range(1,iter+1):
                Elm = rs.coercebrep(rs.AddSrfPt([ptsFT1[i],ptsFT1[i-1],pts
                UnSclMesh.append(Elm)
                Elm = Elm.Duplicate()
                Elm.Transform(Mrr)
                UnSclMesh.append(Elm)
                Elm = rs.coercebrep(rs.AddSrfPt([ptsFT2[i],ptsFT2[i-1],pts
                UnSclMesh.append(Elm)
                Elm = Elm.Duplicate()
                Elm.Transform(Mrr)
                UnSclMesh.append(Elm)
                Elm = rs.coercebrep(rs.AddSrfPt([ptsW2[i-1],ptsW1[i-1],ptsW
                UnSclMesh.append(Elm)
                Elm = Elm.Duplicate()
                Elm.Transform(Mrr)
                UnSclMesh.append(Elm)
        elif iter == 0:
            UnSclMesh.append(rs.coercesurface(rs.AddPlanarSrf(Ln)))
    #Scale and return the section and Mesh:
        if Scaling:
            Scl = rh.Geometry.Transform.Scale(rh.Geometry.Plane.WorldYZ,Sc
        else:
            Scl = rh.Geometry.Transform.Scale(rh.Geometry.Plane.WorldYZ,0.0
        for i in range(0,len(UnSclMesh)):
            UnSclMesh[i].Transform(Scl)
            Mesh.append(UnSclMesh[i])
```

```
Srf = rs.coercebrep(rs.AddPlanarSrf(Ln))
Srf.Transform(Scl)
Section = Srf
```

A.10 Joint

```
import rhinoscriptsyntax as rs
import Rhino as rh
import scriptcontext as sc
import clr
import math
clr.AddReference("Grasshopper")
import Grasshopper.Kernel.Data.GH_Path as ghpath
import Grasshopper.DataTree as datatree
import System
##############################################################
##############################################################
#Create trees
Crvs = datatree[System.Object]()
Elements = datatree[System.Object]()
Geometry = datatree[System.Object]()
Inf = datatree[System.Object]()
Tock = datatree[System.Object]()

#Create Lists
Part = []
Instance = []
Sets = []
BC = []
Tie = []
Steps = []
Vertices = []
Srf = []
List = []

##############################################################
##############################################################
if (Center - Rail.PointAtStart).Length < sc.doc.ModelAbsoluteTolerance
    Ev = 0
    Tngt = Rail.TangentAt(1)
    Direction = Tngt - rh.Geometry.Vector3d(0,0,Tngt.Z)
    rh.Geometry.Vector3d.Unitize(Direction)
    rh.Geometry.Vector3d.Reverse(Direction)
else:
    Ev = 1
```

```
    Tngt = Rail.TangentAt(0)
    Direction = Tngt - rh.Geometry.Vector3d(0,0,Tngt.Z)
    rh.Geometry.Vector3d.Unitize(Direction)
Trns = rh.Geometry.Transform.Translation(-Direction*0.5*PlateThick)
Center.Transform(Trns)
if Tngt.Z - Center.Z == 0:
    StrtPt1 = Center - 0.5*Direction*PlateThick
else:
    Crv = rh.Geometry.Line(Center, -rh.Geometry.Vector3d.ZAxis*Center.
    StrtPt1 = rs.coerce3dpoint(sc.doc.Objects.AddPoint(rh.Geometry.Int
if Ev == 1:
    if Rail.ClosestPoint(StrtPt1)[1]-Length_Memb/Rail.Line.Length < 0:
        EndLn1 = Rail.PointAt(0)
    else:
        EndLn1 = Rail.PointAt(Rail.ClosestPoint(StrtPt1)[1]-Length_Mem
else:
    if Rail.ClosestPoint(StrtPt1)[1]+Length_Memb/Rail.Line.Length > 1:
        EndLn1 = Rail.PointAt(1)
    else:
        EndLn1 = Rail.PointAt(Rail.ClosestPoint(StrtPt1)[1]+Length_Mem
if Ev == 0:
    V1 = (rh.Geometry.Point3d(StrtPt1.X,StrtPt1.Y,0)-rh.Geometry.Point
    V2 = (rh.Geometry.Point3d(EndLn1.X,EndLn1.Y,0)-rh.Geometry.Point3d
else:
    V1 = (rh.Geometry.Point3d(StrtPt1.X,StrtPt1.Y,0)-rh.Geometry.Point
    V2 = (rh.Geometry.Point3d(EndLn1.X,EndLn1.Y,0)-rh.Geometry.Point3d
StrtPt2 = rh.Geometry.Point3d(StrtPt1.X,StrtPt1.Y,StrtPt1.Z)
Trns = rh.Geometry.Transform.Translation(2*Direction*V1)
StrtPt2.Transform(Trns)
EndLn2 = rh.Geometry.Point3d(EndLn1.X,EndLn1.Y,EndLn1.Z)
Trns = rh.Geometry.Transform.Translation(2*Direction*V2)
EndLn2.Transform(Trns)
RailM1 = rh.Geometry.Line(StrtPt1,EndLn1).ToNurbsCurve()
RailM2 = rh.Geometry.Line(StrtPt2,EndLn2).ToNurbsCurve()
#########################################################
#########################################################
#Discretize Plate
EndPt = Center + Direction*PlateThick
RailPl = rs.coercecurve(rs.AddLine(Center,EndPt))
Geometry.Add(RailPl,ghpath(0,0,0))
Pts = rs.DivideCurve(RailPl,1)
```

```
PrPts = RailPl.DivideByCount(1,True)
for i in range(0,2):
    Trns = rs.VectorCreate(Pts[i],rh.Geometry.Vector3d(0,0,0))
    Tngt = RailPl.TangentAt(PrPts[i])
    rot = rh.Geometry.Transform.Rotation(rh.Geometry.Vector3d.XAxis,Tn
    Brds = Mesh_Plate.Branch(0)[0].Duplicate()
    Brds.Translate(Trns)
    Brds.Transform(rot)
    Srf.append(Brds)
    Brds = Brds.DuplicateEdgeCurves()
    for j in range(0,len(Brds)):
        Crvs.Add(Brds[j], ghpath(i))
    for j in range(0,len(Mesh_Plate.Branch(1))):
        Obj = Mesh_Plate.Branch(1)[j].Duplicate()
        Obj.Translate(Trns)
        Obj.Transform(rot)
        Vrt = Obj.DuplicateVertices()
        Lns = Obj.DuplicateEdgeCurves()
        List.append(Obj)
        for k in range(0, len(Lns)):
            Vertices.append(Vrt[k])
            Elements.Add(Vrt[k],ghpath(0,i,j))
            Geometry.Add(Lns[k],ghpath(1,0,i))

if Run == True:
    for i in range(0,len(Crvs.Branch(i))):
        Lft = rs.AddLoftSrf([Crvs.Branch(0)[i],Crvs.Branch(1)[i]],None
        Srf.append(Lft[0])
    Shape = rs.JoinSurfaces(Srf)
    Geometry.Add(Shape,ghpath(0,0,0))
    Part.append('*Part, name={}-Plate'.format(ID))
    Part.append("*Node")
    Nds = rs.CullDuplicatePoints(Vertices)
    del Vertices[:]
    for i in range(0, len(Nds)):
        Nr = str(i+1)
        if ModelOrigin:
            Crd = (Nds[i]-ModelOrigin)
        else:
            Crd = Nds[i]
        X = str(Crd.X)
```

```
        Y = str(Crd.Y)
        Z = str(Crd.Z)
        Coord = [Nr, X, Z, Y]
        TEXT = ',   '.join(Coord)
        Part.append(TEXT)
    Part.append('*Element, type={}'.format(ElmntType[0]))
    Count = 1
    for j in range(0, len(Mesh_Plate.Branch(1))):
        ElNr = [str(Count)]
        for k in range(0, len(Elements.Branch(0,0,j))):
            for l in range(0, len(Nds)):
                if rs.PointCompare(Elements.Branch(0,0,j)[k],Nds[l]):
                    ElNr.append(str(l+1))
        for k in range(0, len(Elements.Branch(0,0,j))):
            for l in range(0, len(Nds)):
                if rs.PointCompare(Elements.Branch(0,1,j)[k],Nds[l]):
                    ElNr.append(str(l+1))
            if rs.PointCompare(Srf[0].ClosestPoint(Elements.Branch(0,0
                Tock.Add(k+1,ghpath(0))
                if len(Tock.Branch(0)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,0,4))
                    Inf.Add(1,ghpath(1,0,5))
            if rs.PointCompare(Srf[0].ClosestPoint(Elements.Branch(0,1
                Tock.Add(k+5,ghpath(n))
                if len(Tock.Branch(0)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,0,4))
                    Inf.Add(1,ghpath(1,0,5))
            if rs.PointCompare(Srf[1].ClosestPoint(Elements.Branch(0,0
                Tock.Add(k+1,ghpath(1))
                if len(Tock.Branch(1)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,1,4))
                    Inf.Add(2,ghpath(1,1,5))
            if rs.PointCompare(Srf[1].ClosestPoint(Elements.Branch(0,1
                Tock.Add(k+5,ghpath(1))
                if len(Tock.Branch(1)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,1,4))
                    Inf.Add(2,ghpath(1,1,5))
        TEXT = ',   '.join(ElNr)
        Part.append(TEXT)
        Count = Count + 1
        Tock.ClearData()
```

```
        del ElNr[:]
    Part.append("*Nset, nset=Set-1, generate")
    Part.append('    1,     {},     1'.format(len(Nds)))
    Part.append("*Elset, elset=Set-1, generate")
    Part.append('    1,     {},     1'.format(Count-1))
    Part.append('**Section: {}_section '.format(ID))
    Part.append('*Solid Section, elset=Set-1, material={}'.format(Mate
    Part.append(",")
    Part.append("*End Part")
    Part.append("**")
    Instance.append('*Instance, name={}-Plate, part={}-Plate'.format(II
    Instance.append("*End Instance")
    Instance.append("**")
    Sets.append('*Nset, nset=Displacement, instance={}-Plate'.format(II
    for i in range(0,2):
        for j in range(0,len(Nds)):
            if rs.PointCompare(Nds[j],Srf[i].ClosestPoint(Nds[j])) ==
                Sets.append(''.join([str(i+1),","]))
    for i in range(0,2):
        Sets.append('*Elset, elset=Tie{}-{}-Plate-TieSrf, internal, in
        for j in range(0,len(Inf.Branch(1,i,4))):
            Sets.append(Inf.Branch(1,i,4)[j])
    for i in range(0, 2):
        Sets.append('*Surface, type=ELEMENT, name=Tie{}-{}-Plate-TieSr
        Sets.append('Tie{}-{}-Plate-TieSrf, S{}'.format(i+1,ID,Inf.Bra
    del Srf[:]
    del ElNr[:]
    Inf.ClearData()
    Elements.ClearData()
    Crvs.ClearData()
##########################################################
##########################################################
#Discretize Member1
Pts = rs.DivideCurve(RailM1,Divisions)
PrPts = Rail.DivideByCount(Divisions,True)
for i in range(0,Divisions+1):
    Trns = rs.VectorCreate(Pts[i],rh.Geometry.Vector3d(0,0,0))
    Tngt = -Direction
    xprd = rs.VectorCrossProduct(Rail.TangentAt(PrPts[i]),Tngt)
    pln = rs.PlaneFromNormal(Pts[i],xprd)
    Scly = 1/math.cos(rh.Geometry.Vector3d.VectorAngle(Rail.TangentAt(
```

```
    if xprd == rh.Geometry.Vector3d(0,0,0):
        Sclz = 1.0
    else:
        Sclz = 1/math.cos(rh.Geometry.Vector3d.VectorAngle(Rail.Tangen
    Scly = 1
    Sclz = 1
    Scle = rh.Geometry.Transform.Scale(rh.Geometry.Plane.WorldYZ,Scly,
    rot = rh.Geometry.Transform.Rotation(rh.Geometry.Vector3d.XAxis,Tn
    Brds = Mesh_Member.Branch(0)[0].Duplicate()
    Brds.Transform(Scle)
    Brds.Translate(Trns)
    Brds.Transform(rot)
    if i == 0 or i == Divisions:
        Srf.append(Brds)
    Brds = rs.DuplicateEdgeCurves(Brds)
    for j in range(0,len(Brds)):
        Crvs.Add(Brds[j], ghpath(i))
    for j in range(0,len(Mesh_Member.Branch(1))):
        Obj = Mesh_Member.Branch(1)[j].Duplicate()
        Obj.Transform(Scle)
        Obj.Translate(Trns)
        Obj.Transform(rot)
        Vrt = Obj.DuplicateVertices()
        Lns = Obj.DuplicateEdgeCurves()
        List.append(Obj)
        for k in range(0, len(Lns)):
            Vertices.append(Vrt[k])
            Elements.Add(Vrt[k],ghpath(0,i,j))
            Geometry.Add(Lns[k],ghpath(1,0,i))

if Run == True:
    for i in range(0,Divisions):
        for j in range(0,len(Crvs.Branch(i))):
            Lft = rs.AddLoftSrf([Crvs.Branch(i)[j],Crvs.Branch(i+1)[j]
            Srf.append(Lft[0])
    Shape = rs.JoinSurfaces(Srf)
    Geometry.Add(Shape,ghpath(0,0,0))
    Part.append('*Part, name={}-Member1'.format(ID))
    Part.append("*Node")
    Nds = rs.CullDuplicatePoints(Vertices)
    del Vertices[:]
```

```
for i in range(0, len(Nds)):
    Nr = str(i+1)
    if ModelOrigin:
        Crd = (Nds[i]-ModelOrigin)
    else:
        Crd = Nds[i]
    X = str(Crd.X)
    Y = str(Crd.Y)
    Z = str(Crd.Z)
    Coord = [Nr, X, Z, Y]
    TEXT = ',   '.join(Coord)
    Part.append(TEXT)
Part.append('*Element, type={}'.format(ElmntType[0]))
Count = 1
for i in range(0, Divisions):
    for j in range(0, len(Mesh_Member.Branch(1))):
        ElNr = [str(Count)]
        for k in range(0, len(Elements.Branch(0,i,j))):
            for l in range(0, len(Nds)):
                if rs.PointCompare(Elements.Branch(0,i,j)[k],Nds[l
                    ElNr.append(str(l+1))
        for k in range(0, len(Elements.Branch(0,i,j))):
            for l in range(0, len(Nds)):
                if rs.PointCompare(Elements.Branch(0,i+1,j)[k],Nds
                    ElNr.append(str(l+1))
            if rs.PointCompare(Srf[0].ClosestPoint(Elements.Branch
                Tock.Add(k+1,ghpath(0))
                if len(Tock.Branch(0)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,0,4
                    Inf.Add(1,ghpath(1,0,5))
            if rs.PointCompare(Srf[0].ClosestPoint(Elements.Branch
                Tock.Add(k+5,ghpath(n))
                if len(Tock.Branch(0)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,0,4
                    Inf.Add(1,ghpath(1,0,5))
        TEXT = ',   '.join(ElNr)
        Part.append(TEXT)
        Count = Count + 1
        Tock.ClearData()
        del ElNr[:]
Part.append("*Nset, nset=Set-1, generate")
```

```
        Part.append('    1,     {},     1'.format(len(Nds)))
        Part.append("*Elset, elset=Set-1, generate")
        Part.append('    1,     {},     1'.format(Count-1))
        Part.append('**Section: {}_section '.format(ID))
        Part.append('*Solid Section, elset=Set-1, material={}'.format(Mate
        Part.append(",")
        Part.append("*End Part")
        Part.append("**")
        Instance.append('*Instance, name={}-Member1, part={}-Member1'.form
        Instance.append("*End Instance")
        Instance.append("**")
        Sets.append('*Nset, nset=Sup-Member1, instance={}-Member1'.format(
        for i in range(0,len(Nds)):
            if rs.PointCompare(Nds[i],Srf[1].ClosestPoint(Nds[i])) == True
                Sets.append(''.join([str(i+1),","]))
        Sets.append('*Elset, elset=Tie1-{}-Member1-TieSrf, internal, instan
        for j in range(0,len(Inf.Branch(1,0,4))):
            Sets.append(Inf.Branch(1,0,4)[j])
        Sets.append('*Surface, type=ELEMENT, name=Tie1-{}-Member1-TieSrf'.
        Sets.append('Tie1-{}-Member1-TieSrf, S{}'.format(ID,Inf.Branch(1,0
        BC.append("Sup-Member1, 1, 1")
        BC.append("Sup-Member1, 2, 2")
        BC.append("Sup-Member1, 3, 3")
        del Srf[:]
        del ElNr[:]
        Inf.ClearData()
        Elements.ClearData()
        Crvs.ClearData()
##############################################################
##############################################################
#Discretize Member2
Pts = rs.DivideCurve(RailM2,Divisions)
PrPts = Rail.DivideByCount(Divisions,True)
for i in range(0,Divisions+1):
    Trns = rs.VectorCreate(Pts[i],rh.Geometry.Vector3d(0,0,0))
    Tngt = Direction
    xprd = rs.VectorCrossProduct(Rail.TangentAt(PrPts[i]),Tngt)
    pln = rs.PlaneFromNormal(Pts[i],xprd)
    Scly = 1/math.cos(rh.Geometry.Vector3d.VectorAngle(Rail.TangentAt(
    if xprd == rh.Geometry.Vector3d(0,0,0):
        Sclz = 1.0
```

```
        else:
            Sclz = 1/math.cos(rh.Geometry.Vector3d.VectorAngle(Rail.Tangen
        Scly = 1
        Sclz = 1
        Scle = rh.Geometry.Transform.Scale(rh.Geometry.Plane.WorldYZ,Scly,
        rot = rh.Geometry.Transform.Rotation(rh.Geometry.Vector3d.XAxis,Tn
        Brds = Mesh_Member.Branch(0)[0].Duplicate()
        Brds.Transform(Scle)
        Brds.Translate(Trns)
        Brds.Transform(rot)
        if i == 0 or i == Divisions:
            Srf.append(Brds)
        Brds = rs.DuplicateEdgeCurves(Brds)
        for j in range(0,len(Brds)):
            Crvs.Add(Brds[j], ghpath(i))
        for j in range(0,len(Mesh_Member.Branch(1))):
            Obj = Mesh_Member.Branch(1)[j].Duplicate()
            Obj.Transform(Scle)
            Obj.Translate(Trns)
            Obj.Transform(rot)
            Vrt = Obj.DuplicateVertices()
            Lns = Obj.DuplicateEdgeCurves()
            List.append(Obj)
            for k in range(0, len(Lns)):
                Vertices.append(Vrt[k])
                Elements.Add(Vrt[k],ghpath(0,i,j))
                Geometry.Add(Lns[k],ghpath(1,0,i))

if Run == True:
    for i in range(0,Divisions):
        for j in range(0,len(Crvs.Branch(i))):
            Lft = rs.AddLoftSrf([Crvs.Branch(i)[j],Crvs.Branch(i+1)[j]
            Srf.append(Lft[0])
    Shape = rs.JoinSurfaces(Srf)
    Geometry.Add(Shape,ghpath(0,0,0))
    Part.append('*Part, name={}-Member2'.format(ID))
    Part.append("*Node")
    Nds = rs.CullDuplicatePoints(Vertices)
    del Vertices
    for i in range(0, len(Nds)):
        Nr = str(i+1)
```

```
    if ModelOrigin:
        Crd = (Nds[i]-ModelOrigin)
    else:
        Crd = Nds[i]
    X = str(Crd.X)
    Y = str(Crd.Y)
    Z = str(Crd.Z)
    Coord = [Nr, X, Z, Y]
    TEXT = ',   '.join(Coord)
    Part.append(TEXT)
Part.append('*Element, type={}'.format(ElmntType[0]))
Count = 1
for i in range(0, Divisions):
    for j in range(0, len(Mesh_Member.Branch(1))):
        ElNr = [str(Count)]
        for k in range(0, len(Elements.Branch(0,i,j))):
            for l in range(0, len(Nds)):
                if rs.PointCompare(Elements.Branch(0,i,j)[k],Nds[l
                    ElNr.append(str(l+1))
        for k in range(0, len(Elements.Branch(0,i,j))):
            for l in range(0, len(Nds)):
                if rs.PointCompare(Elements.Branch(0,i+1,j)[k],Nds
                    ElNr.append(str(l+1))
            if rs.PointCompare(Srf[0].ClosestPoint(Elements.Branch
                Tock.Add(k+1,ghpath(0))
                if len(Tock.Branch(0)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,0,4
                    Inf.Add(1,ghpath(1,0,5))
            if rs.PointCompare(Srf[0].ClosestPoint(Elements.Branch
                Tock.Add(k+5,ghpath(n))
                if len(Tock.Branch(0)) == 4:
                    Inf.Add(''.join([str(Count),","]),ghpath(1,0,4
                    Inf.Add(1,ghpath(1,0,5))
        TEXT = ',   '.join(ElNr)
        Part.append(TEXT)
        Count = Count + 1
        Tock.ClearData()
        del ElNr[:]
Part.append("*Nset, nset=Set-1, generate")
Part.append('    1,     {},     1'.format(len(Nds)))
Part.append("*Elset, elset=Set-1, generate")
```

```
Part.append('    1,     {},     1'.format(Count-1))
Part.append('**Section: {}_section '.format(ID))
Part.append('*Solid Section, elset=Set-1, material={}'.format(Mate
Part.append(",")
Part.append("*End Part")
Part.append("**")
Instance.append('*Instance, name={}-Member2, part={}-Member2'.form
Instance.append("*End Instance")
Instance.append("**")
Sets.append('*Nset, nset=Sup-Member2, instance={}-Member2'.format(
for i in range(0,len(Nds)):
    if rs.PointCompare(Nds[i],Srf[1].ClosestPoint(Nds[i])) == True
        Sets.append(''.join([str(i+1),","]))
        Geometry.Add(Nds[i].X,ghpath(2,j,0))
        Geometry.Add(Nds[i].Y,ghpath(2,j,1))
        Geometry.Add(Nds[i].Z,ghpath(2,j,2))
BC.append("Sup-Member2, 1, 1")
BC.append("Sup-Member2, 2, 2")
BC.append("Sup-Member2, 3, 3")
Sets.append('*Elset, elset=Tie2-{}-Member2-TieSrf, internal, instan
for j in range(0,len(Inf.Branch(1,0,4))):
    Sets.append(Inf.Branch(1,0,4)[j])
Sets.append('*Surface, type=ELEMENT, name=Tie2-{}-Member2-TieSrf'.
Sets.append('Tie2-{}-Member2-TieSrf, S{}'.format(ID,Inf.Branch(1,0
Tie.append("** Constraint: Tie1")
Tie.append("*Tie, name=Tie1, adjust=yes, type=SURFACE TO SURFACE")
Tie.append('Tie1-{}-Member1-TieSrf, Tie1-{}-Plate-TieSrf'.format(II
Tie.append("** Constraint: Tie2")
Tie.append("*Tie, name=Tie2, adjust=yes, type=SURFACE TO SURFACE")
Tie.append('Tie2-{}-Member2-TieSrf, Tie2-{}-Plate-TieSrf'.format(II
Steps.append("**")
Steps.append("** STEP: Displacement")
Steps.append("**")
Steps.append("*Step, name=Displacement, nlgeom=NO")
Steps.append("*Static")
Steps.append("0.1, 1., 1e-05, 1.")
Steps.append("**")
Steps.append("** BOUNDARY CONDITIONS")
Steps.append("**")
Steps.append("** Name: Displacement Type: Displacement/Rotation")
Steps.append("*Boundary")
```

```
if Tx:
    Steps.append('Displacement, 1, 1, {}'.format(Tx))
if Tz:
    Steps.append('Displacement, 2, 2, {}'.format(Tz))
if Ty:
    Steps.append('Displacement, 3, 3, {}'.format(Ty))
Steps.append("**")
Steps.append("** OUTPUT REQUESTS")
Steps.append("**")
Steps.append("*Restart, write, frequency=0")
Steps.append("**")
Steps.append("** FIELD OUTPUT: F-Output-1")
Steps.append("**")
Steps.append("*Output, field, variable=PRESELECT")
Steps.append("**")
Steps.append("** HISTORY OUTPUT: H-Output-1")
Steps.append("**")
Steps.append("*Output, history, variable=PRESELECT")
Steps.append("*End Step")
del Srf
del ElNr
del Inf
del Elements
del Crvs
```

B Grasshopper Files

B.1 Case1_VonMises.gh - Attachment

B.2 Case2_VonMises.gh - Attachment

B.3 Case3_1_VonMises.gh - Attachment

B.4 Case3_2_VonMises.gh - Attachment

B.5 Case3_3_VonMises.gh - Attachment

B.6 Case3_4_VonMises.gh - Attachment

B.7 Components.gh - Attachment

B.8 GableTruss.gh - Attachment

B.9 Connection.gh - Attachment